APPLICATION OF HALSTEAD'S TIMING MODEL TO PREDICT THE COMPILATION TIME OF ADA COMPILERS

THESIS

AFIT/GE/ENG/86D-7 Dennis M. Miller
Captain USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/86D-7

APPLICATION OF HALSTEAD'S TIMING MODEL TO PREDICT THE COMPILATION TIME OF ADA COMPILERS

THESIS

AFIT/GE/ENG/86D-7 Dennis M. Miller
Captain USAF

AFIT/GE/ENG/86D-7

APPLICATION OF HALSTEAD'S TIMING MODEL TO PREDICT THE COMPILATION TIME OF ADA COMPILERS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements of the Degree of
Master of Science

Dennis M. Miller, B.S.
Captain, USAF

December 1986

Preface

The purpose of this study was to determine the applicability of using Halstead's Software Science theory to predict compile time and evaluate compilers for Ada. The need for more objective tools in evaluating software is becoming more apparent. I think the results of this project are useful and will serve as a baseline for future comparisons. It may be the tool that many researchers, managers, and evaluators have been searching for.

I wish to thank a number of people who helped me complete this research project. In particular, Dr. Wade Shaw, my advisor, and Dr. Jim Howatt both of whom reviewed this work during its development and provided countless helpful suggestions. Without Dr. Howatt's assistance, I would have still been counting operators and operands in Ada programs. Deep gratitude is expressed to Dr. Shaw, who was instrumental in providing the guidance and directions I needed to finish my research effort. I would like to give a special thanks to Captain Deese of the Information Systems and Technology Center at Wright-Patterson AFB, Ohio, who found time in his busy schedule to give me information and instructions on the Data General computer. I am also indebted to Captain Robert S. Maness; a fellow AFIT student, my friend and partner; for his support and patience in getting the research data necessary to complete our respective projects. Finally, I can not forgot the most important person in my life, my wife to be, Amy Bass. Her

patience and continuing support to me deserves more than the usual amount of credit. The encouragement I received from her was inspirational. I couldn't have done it without her.

Table of Contents

		Page
Preface		ii
List of Figures		vi
List of Tables		vii
List of Acronyms		viii
List of Symbols		ix
Abstract		x
I.	Introduction	1
	Background	3
	Problem	6
	Scope	7
	General Assumptions	8
	General Approach	9
	Sequence of Presentation	10
II.	Halstead's Software Science Theory and Its Application to Compilers	12
	Halstead's Software Science Theory	12
	Review of Counting Strategies	19
	Acceptance/Criticisms of Halstead's Work	22
	An Application of Software Science to Compilers	25
III.	Research Methodology	31
	Model Proposals	31
	Model 1 - Time	32
	Model 2 - Length (Linear)	33
	Model 3 - Length (Non-Linear)	34
	The Experiment Design	35
	Data Selection	36
	Identification/Enumeration of Operands, Operators, and I/O Parameters	37
	Computer/Compiler Selection	40
	Computer Environment	41
	Compiler Time Measurements	43
	Unix Environment	46
	AOS/VS Environment	47
	VMS Environment	49
	Statistical Analysis	50
IV.	Test Results and Discussion	55

V. Conclusions and Recommendations..... 70

Conclusions..... 71
Recommendations..... 73

Appendix A: Different Counting Strategies..... 76

Appendix B: Ada Programs to Demonstrate Counting..... 78

Appendix C: Sample Data Sheet..... 82

Appendix D: Data for the Compile Time Study..... 84

Appendix E: Where to Begin..... 88

Appendix F: Actual Compile Times..... 89

Appendix G: Macros Used in the Experiment..... 93

Appendix H: SAS Data File for Analysis 1..... 95

Appendix I: SAS Command Files for Analysis 1..... 96

Appendix J: SAS Data File for Analysis 2..... 98

Appendix K: SAS Command Files for Analysis 2..... 99

Appendix L: Sample SAS Output..... 100

Appendix M: Actual vs Predicted Compile Times..... 104

Appendix N: Plot of the Actual vs Predicted Compile Times..... 108

Bibliography..... 111

Vita..... 114

List of Figures

Figure		Page
2.1	Components of a Generalized Compiler.....	27
3.1	ACEC Compilation Order.....	44
4.1	Compiler Model Comparisons.....	56
4.2	Halstead Model vs Compiler Model.....	60
4.3	UNIX Compile Time: Actual vs Model Prediction.....	63
4.4	AOS/VS Compile Time: Actual vs Model Prediction.....	64
4.5	VMS-JSL Compile Time: Actual vs Model Prediction.....	65
4.6	VMS-CSC Compile Time: Actual vs Model Prediction.....	66
4.7	CPU Performance Comparison.....	69

List of Tables

Table		Page
2.1	Halstead's Software Science Measures.....	13
3.1	Ada Counting Strategy.....	38
4.1	Mathematical Models for Compile Times.....	55
4.2	Error Reduction in Predicting Compile Time...	57
4.3	Parameter Estimates.....	58
4.4	Correlation Between Observed and Predicted Compile Times.....	61
4.5	Residual Error Comparison.....	62
4.6	Parameter Estimates for Pooled Data.....	67
4.7	Compiler Evaluation.....	68
5.1	Correlation Between Actual and Predicted Compilation Times.....	71

List of Acronyms

ACEC	Ada Compiler Evaluation Capability
ADE	Ada Development Environment
AFIT	Air Force Institute of Technology
ANSI	American National Standards Institute
ASC	Academic Support Computer
ASD	Aeronautical System Division
bpi	Bits per Inch
CPU	Central Processor Unit
CSC	Classroom Support Computer
DEC	Digital Equipment Corporation
DG	Data General
DoD	Department of Defense
E & V	Evaluation and Validation
IDA	Institute for Defense Analysis
I/O	Input/Output
lpm	Lines per Minute
ISL	Information System Laboratory
LRM	Language Reference Manual
MIL-STD	Mitilary Standard
RTS	Real-Time Support

List of Symbols

E - Effort

L - Level of Implementation

Lhat - Estimated Level of Implementation

n_1 - Total Number of Unique Operators

n_2 - Total Number of Unique Operands

n - Vocabulary

n_2^* - Input and output parameters

N_1 - Total Number of Operators

N_2 - Total Number of Operands

N - Length of a Program

Nhat - Estimated Length of a Program

S - Stroud Number

T - Time

V - Volume

Vhat - Estimated Volume

V^* - Potential Volume

Abstract

With the development of Ada, the official programming language of the DoD, methods are needed to validate and evaluate various Ada compilers to determine if the compilers meet the DoD requirements. This ~~investigation~~ thesis introduces a new tool using Halstead's Software Science theory to predict compile time and to evaluate the efficiency of alternative Ada compilers.

The analysis was accomplished by selecting a model based on Halstead's time equation. Once the model was established, programs from a benchmark test suite were used to evaluate the predictive power of the model and to develop a performance index for comparisons.

The results suggest that the compiler model is useful in predicting compile time, but of more importance, it is useful in the development of a performance index. The study shows that the average compilation time is not a good measure for comparing performance rates. Therefore, with further research, the compiler model may be a useful tool for software analysts.

APPLICATION OF HALSTEAD'S TIMING MODEL TO PREDICT THE COMPILATION TIME OF ADA COMPILERS

I. Introduction

Technological advances in computer software are changing our society. Computer systems are becoming more numerous, more complex, and deeply embedded in our society. We can no longer simply write programs, but must engineer software for our systems to help offset the rising cost of software development (9:8-9). The Department of Defense (DoD) recognized this challenge and realized that a new standard language and its environment (i.e. compilers, loaders, library managers, etc) had to be created to encourage the use of modern software engineering techniques (4). As a result, the Ada programming language was developed. With the introduction of this new standard programming language for the DoD, software engineering tools are needed to evaluate the performance and reliability of programs written in this language. These tools are more important today because millions of dollars of equipment, and even lives may depend on the proper execution of these computer programs.

Ada (4:1-21) was developed under sponsorship of the DoD to support the development of software for embedded computer systems. For example, one area of use will be in the field of avionics. In the development of avionics software,

efficient compilers are needed. Therefore, new tools, besides benchmark test suites, are needed to evaluate compiler performance (good code, optimization, compilation time, etc). Benchmark test suites have a bad reputation because the performance figures are sometimes cited out of context and overgeneralized into overall ratings (20:31). One possible new tool for determining a performance index for compilation time in compilers is to use an extension of Maurice Halstead's Software Science theory.

Maurice Halstead developed a theory called Software Science with the objective of making sound judgements about software quality and complexity. Software Science theory (9:13) is based on the fact that algorithms can be measured by their physical characteristics, i.e. the number of distinct operators and operands and the total number of operators and operands within the computer program. Using this assumption, Halstead was able to develop several mathematical formulas that accurately express several attributes of algorithms. One of the formulas, the programmer time equation, developed from Halstead's study can be used to express the amount of time required for a programmer to translate a predefined algorithm into a given computer programming language.

It is interesting to speculate about other uses of Halstead's formulas. For instance, the human translation of an algorithm into a programming language can be considered to be similar to the process that a compiler goes through to

translate a computer program into executable machine code. Given that Halstead's formula can predict the time required for the human translation process, it is interesting to speculate if it can be used to accurately predict the time required for compilation of a program. If this can be done, performance evaluation can be performed on compilers to determine their efficiency. Consequently, software science may be a possible tool for analyzing compilation times.

This paper describes a research effort to determine if an extension of Halstead's theory on predicting time is applicable to Ada compilers, and thereby able to provide a performance index for comparisons. If so, objective decisions on at least one metric of compiler performance is possible.

Background

In the 1970's, the DoD (4:1-21) recognized a need for a standard, high-order language to reduce the cost and effort to develop and maintain military software systems. However, a suitable language did not exist that met all the requirements. As a result, the DoD sponsored a development effort to produce a new language which has become known has Ada, after Lady Augusta Ada Byron, the world's first programmer.

According to Booch (4:44), Ada is a strongly typed language that provides a rich set of constructs for describing primitive objects and operations, and in

addition, offers a packaging construct with which we may build and enforce our own abstraction. Features such as exception handling, parallel processing, real-time control, and information hiding, makes Ada a language useful for many diverse applications.

The various modern language features incorporated in Ada are intended to improve software quality and increase programmer productivity. The language seems promising. However, for Ada (25) to serve as the official language for DoD, compilers need to be developed which conform to the Ada language specifications and produce efficient object code. As a result, methods are needed to validate and evaluate various compilers in order to determine which could best meet DoD requirements.

In validating and evaluating compilers, performance information such as compilation time, memory space requirements, object code generation, error checking, etc must be analyzed. Currently, benchmark performance test suites are used. For example, the Ada Evaluation and Validation (E&V) team collected numerous test routines from the public domain to provide users with "1) an organized suite of compiler performance tests, and 2) support software for executing these tests and collecting performance statistics" (12). This test suite is called the Prototype Ada Compiler Evaluation Capability (ACEC). Currently, the ACEC is not completed; it fails to test all the features of the language.

Benchmark test suites are useful in evaluating compilers, but tests must be complete and repeatable. The ACEC measurements, for example, "... are only an indication of the effect produced by an Ada language feature when it is used in a particular compiler/run-time combination. These measurements are not absolute performance metrics of the efficiency of a particular compiler architecture" (12). Benchmark test suites have a bad reputation because the tests are sometimes misapplied, incorrectly performed, or inadequately documented (20). Therefore, other methods are needed to make objective decisions in evaluating compilers. For example, the performance metric, compilation time, is a perfect example to use in the development of a mathematical model to determine the performance of various compilers. The author is not suggesting to replace benchmark test suites for evaluating compilers; however, using an extension of Halstead's Software Science theory may provide evaluators the tool to make consistent and objective evaluations of at least one performance metric - compilation time. This model could be more useful than using the average compilation time to evaluate compilers.

According to Halstead (11:3), Software Science is defined as follows:

> Software Science is concerned with algorithms and their implementation, either as computer programs, or as instruments of human communication. As an experimental science, it deals only with those properties of algorithms that can be measured, either directly or indirectly, statically or dynamically, and with the relationships among those properties that remain invariant under translation from one language to another.

Halstead (10:4) undertook his study on the properties of algorithms (computer programs) with the objective of making quality judgments about the size and the programming effort required to create them. Specifically, he was interested in predicting the time and effort required for a programmer to write a program, the length of the program, and the number of programming errors generated. He developed a theoretical framework based on the number of operands and operators in a algorithm and demonstrated that the theory can be validated (14:13-35). A detailed discussion on Halstead's Software Science formulas is presented in Chapter II. The question now is - can Halstead's model of programming time be used for compilers?

Problem

The problem addressed in this thesis is to investigate the predictive power of Halstead's model of Software Science in estimating compilation time across alternative Ada compilers.

Scope

This study concentrates on the compilation process, applying concepts developed by software science. Since Ada is the new DoD standard for programming languages and is of high interest in the military community, this thesis focuses on Ada compilers.

The purpose of this investigation is two-fold: (1) To determine if there is significant difference in the predictive ability of Halstead's model of Software Science in explaining compilation time among alternative Ada compilers; and (2) To determine if there is significant difference in the discrimination rate across alternative Ada compilers. With this in mind, this thesis will:

(1) Develop Halstead's Software Science theory and its application to compilers.

(2) Develop a counting strategy for Ada. select a set of Ada programs, and select a set of Ada compilers.

(3) Design a statistical model and performance index.

(4) Analyze the model, test the hypotheses, and summarize the results.

General Assumptions

Compilation time is influenced by many factors. For instance, how a compiler is written will affect the compilation effort - one-pass, two-pass, and/or optimized or not, etc. Halstead's mathematical model for programming effort is based on properties of the programming language, not on the ability of the programmer. It seems reasonable to approach the compilation effort in a similar fashion. Correlation of data from this study with the theoretical estimates is used to justify the extension of Halstead's model in predicting compilation time. The justification for this assumption is that Halstead's model performs well (11:46-61) in predicting the time for a programmer to translate an algorithm from a mathematical model into some high order language. It then might be assumed to be a good model for estimating compiler time, since a compiler is performing the same function as a programmer - translating an algorithm from one level language to another.

For the purpose of this study, all compilers examined have been validated, all programs used compile correctly, and all compilation times are the results of no optimization. Additionally, the discrimination rate Halstead used to describe the programmer speed is assumed to be the translation or processing rate for a compiler.

Although this study does not cover compiler design, properties of algorithms, or different languages, success of this investigation might generate further experiments designed to test the extension of Halstead's model for predicting the time for the compiler to translate a program from one language to another.

General Approach

Halstead's model of Software Science was used to propose a general model for predicting compilation time. An experiment was designed to collect data. This data was used to estimate the unknown variables of the mathematical model and to test relevant hypotheses.

The experimental design required that the algorithms be selected with a wide range of software science metrics. The algorithms were compiled on four different computers having Ada compilers and the compilation times were recorded. A major issue was measuring the compile time as accurately as possible. On a multi-user computer system, compilation time cannot be measured simply by a stop watch because of the contention with other users. Therefore, total CPU time used in the compilation process was used. This time was obtained from the list or history file generated by the compiler.

The software metrics necessary for the proposed mathematical model were extracted from the algorithms manually. This required a set of rules for the identification and enumeration of each operator, operand,

and I/O variables in each program. A program was measured by applying the counting rules; and then, based on the resulting parameter values, the various software metrics were calculated. At this point, several mathematical models based on software science metrics were proposed in predicting compilation time for a compiler. The models were then evaluated using the analysis of variance method and the linear regression tool on the SAS software package for data analysis. Based on this evaluation, one model was selected for further analysis.

The model was used to test two hypotheses:

(1) There is no significant difference in the predictive ability of Software Science in explaining compile time across alternative Ada compilers.

(2) There is no significant difference in the discrimination rate across alternative Ada compilers.

The correlation, or lack of correlation, of the estimates with the actual compilation time will indicate the merit of using Halstead's Software Science theory in predicting the time to compile an Ada algorithm. If there is a correlation between software science and compilation time, then the development of a performance index may become a valuable tool for DoD, in validating and evaluating Ada compilers.

Sequence of Presentation

Chapter II provides an overview of Halstead's Software Science theory. Since software science is based on the operators and operands of a software program, a discussion on counting strategies is given. In addition, a review of published findings covering both acceptance and criticisms is presented. Finally, why software science can be used in explaining compilation time for compilers is discussed. Chapter II was written with the cooperation of Captain Robert S. Maness (17), whose thesis validated the use of software science in explaining compiler time.

In Chapter III, an explanation of the research methodology used to evaluate Halstead's Software Science to analyze compiler time is presented. Chapter IV, contains the results of the experiment. Finally, Chapter V, the conclusions and recommendations, summarizes the results, describes the worthiness of the compiler prediction model, and recommends areas for further study.

II. A Review of Halstead's Software Science Theory and Its Application to Compilers

Maurice Halstead, in his classic work on Software Science (11), attempted to define and measure the complexity of software by using mathematical models. With these mathematical models, Halstead was able to predict software engineering metrics such as the number of errors in a program, the programmer's time for implementation, and the difficulty of implementing a program. The theory's accuracy in predictions has been shown to be both adequate and inadequate (10; 11; 23).

The first section in this chapter presents the theory applicable to this investigation to provide a background for the model to be presented in Chapter III. The second section reviews different counting strategies. In the third section, the acceptance and criticisms of Halstead's work are discussed. Finally, the last section describes the application of Software Science theory to compilers.

The Theory of Software Science

Software science was developed to measure the properties of algorithms. Halstead (11:5-6) defined four basic metrics that are capable of being counted or measured:

n_1 = the number of unique operators; (2.1)
n_2 = the number of unique operands; (2.2)
N_1 = the total number of occurrences of operators; (2.3)
N_2 = the total number of occurrences of operands. (2.4)

According to Halstead, operands are defined as the variables or constants that the implementation employs. While operators are classified as the symbols or combinations of symbols, such as mathematical symbols, delimiters, punctuation symbols, et cetera that affect the value or ordering of an operand (11:5). By counting the number of operators and operands or tokens in a program, software science attempts to measure the programming requirements, the initial error rates, the quality and the complexity of software, and the productivity of programmers (10:3-5; 11). Table 1 summarizes Halstead's measures which are relevant to this study.

TABLE 2.1

Halstead's Software Science Measures (11:2)

(1)	n_1 = Unique Operators
(2)	n_2 = Unique Operands
(3)	Vocabulary = $n = n_1 + n_2$
(4)	N_1 = Total Operators
(5)	N_2 = Total Operands
(6)	Length = $N = N_1 + N_2$
(7)	Est. Length = $\hat{N} = (n_1 * \log_2(n_1)) + (n_2 * \log_2(n_2))$
(8)	Volume = $V = N * \log_2(n)$
(9)	Est. Volume = $\hat{V} = \hat{N} * \log_2(n)$
(10)	Potential Volume = $V^* = (2+n_2^*)*\log_2(2+n_2^*)$
(11)	Level of Implementation = $L = V^* / V$
(12)	Est. Level = $\hat{L} = (2 * n_2) / (n_1 * N_2)$
(13)	Effort = $E = V / L = V^2 / V^*$
(14)	Programming Time = $T = E / S = V^2 / (S * V^*)$

Using the basic metrics above, Halstead (10:5; 11:6) defined the vocabulary n of a program to be the total number of unique tokens:

$$n = n_1 + n_2, \tag{2.5}$$

and the length of a program to be the total number of operators and operands:

$$N = N_1 + N_2. \tag{2.6}$$

Halstead (10:6) hypothesized that the length of a program is a function only of the number of unique operators and operands. Other characteristics of a program are defined using these basic terms. Drawing on intuition, Halstead (2:774) used an analytical procedure and a probability model of software generation to predict the length of a program. Halstead determined that as a program with n unique and N total operators grows in size, by increasing the number of unique tokens, the total length will grow logarithmicly: $n*\log_2 n$. Based on this conclusion, and that the length of a program is the sum of the operators and operands, Halstead (10:5-6; 11:9-11) defined the predicted length or the length estimator as:

$$Nhat = (n_1 * \log_2 n_1) + (n_2 * \log_2 n_2). \tag{2.7}$$

The size or volume of a program may vary when translating from one language to another. For example, converting a higher level language such as Ada into a lower level implementation code (machine language) requires more volume than translating a lower level language into a higher level language. Higher level languages usually have more operators to allow for more compact expressions; and as a result, shorter programs. Halstead (10:6-8; 11:19; 23:156) surmised that the volume of a program is a function of its vocabulary and is given by:

$$V = N * \log_2(n), \tag{2.8}$$

where V has a unit of measurement in bits. In other words, $\log_2(n)$ bits are needed for each of the N tokens in a program to choose one of the operators or operands for that token.

Programs may be implemented by many different but equivalent codes. When an algorithm is implemented in its most succinct form, then its potential volume V^* (11:20-21; 23:156) is

$$V^* = (2 + n_2^*) * \log_2(2 + n_2^*), \tag{2.9}$$

where n_2^* is the number of input/output (I/O) parameters. This represents the size of the program if it existed as a built-in function or procedure call. The constant 2

represents the minimum number of operators for any algorithm to perform the function. One operator is the name of the function or procedure and the other is an assignment or grouping symbol used to separate the list of parameters from the function or procedure name. The variable n_2^* is the minimum number of unique operands (I/O parameters) needed to implement the function. The value for n_2^* is harder to obtain because what constitutes an I/O parameter may be difficult to conceptualize. Halstead describes n_2^* as follows:

> (1) The number of conceptually unique arguments and results (or input and output parameters) required by a given algorithm. Therefore, it is only necessary to count the parameters listed in a call when an algorithm is implemented as a simple procedure, or as a subroutine, and for which a call on that procedure has been written, and provided that result operand names are listed explicitly.
>
> (2) For the cases in which an algorithm is implemented as a straight routine to be executed directly, n_2^* is determined by examining the implementation and by counting all the operands that are "busy-on-entry" or "busy-on-exit" of an algorithm from the implementation. (11:28)

According to Halstead (10:8-9; 11:25-30; 23:156), the level of implementation is defined as the ratio of potential to actual volume:

$$L = V^* / V, \tag{2.10}$$

where L is less than or equal to one. The closer the volume V is to the potential volume V^*, the higher the level. The higher level languages such as Ada should have a value

closer to one than a lower level language because the lower level language usually requires more operators and operands to do the same job. Note also that the failure to use a language properly could result in a lower level of implementation and a higher volume.

Halstead (10:9-10; 11:46-61) hypothesized that a program is generated by making $N * \log_2(n)$ mental comparisons. Therefore the volume is a count of the number of mental comparisons required to generate a program. Each mental comparison requires a number of elementary mental discriminations which are defined as the reciprocal of the level of implementation - 1/L. Halstead then concluded that the total number of elementary mental discriminations or effort E required to generate an algorithm is given as:

$$E = V / L. \tag{2.11}$$

The effort of programming increases as the volume of the program increases and the effort decreases as the level of implementation increases. In other words, the larger a program, the more difficult the effort; the higher the level of implementation, the easier the effort. Recalling Equation 2.10, $L = V^* / V$ and substituting in Equation 2.11, the effort equation now becomes:

$$E = V^2 / V^*. \tag{2.12}$$

Equation 2.12 indicates that the effort required to generate a program with a given potential volume varies with the square of the actual volume in any language. With this equation, Halstead determined the number of mental discriminations or decisions completed by a programmer when implementing an algorithm.

As stated in the introduction, a major claim of software science is the ability to predict actual programming time. Halstead (10:9-10; 11:46-61; 23:157) determined that the amount of time required to implement an algorithm is directly proportional to the programming effort E divided by a constant 'S'.

$$T = E / S,$$

or

$$T = V^2 / (S * V^2), \qquad (2.13)$$

where the constant 'S' represents the speed of the programmer or the number of mental discriminations per second of which the programmer is capable. Halstead (10:9-10; 11:48-49; 23:157) called 'S' the "Stroud number" because a psychologist, J. Stroud proposed that the human brain is able to make mental discriminations at a finite rate (between 5 and 20). Halstead uses a value of 18 because in his experiments, 18 gave him the best results when comparing predicted versus actual programming time. Software science hypothesizes that Equation 2.13 estimates the time required

for a programmer to implement an algorithm under certain conditions (23:157):

(1) A single, concentrating programmer, who is knowledgeable of the programming language;

(2) Only a single module is written;

and (3) The program must be pure (10:6; 11:38-45).

Good programming practices usually insures a pure program. Halstead defined six impurity classes:

1. CANCELLING of OPERATORS: The occurrence of an inverse cancels the effect of a previous operator; no other use of the variable changed by the operator is made before the cancellation.
2. AMBIGUOUS OPERANDS: The same operand is used to represent two or more variables in an algorithm.
3. SYNONYMOUS OPERANDS: Two or more operand names represent the same variable.
4. COMMON SUBEXPRESSION: The same subexpression occurs more than once.
5. UNNECESSARY REPLACEMENTS: A subexpression is assigned to a temporary variable which is used only once.
6. UNFACTORED EXPRESSIONS: There are repetitions of operators and operands among unfactored terms in an expression. (10:6)

Review of Counting Strategies

Since Halstead's theory is based on the counting of operators and operands within a program, a discussion of the method of recognizing and categorizing these tokens is appropriate. As stated before, Halstead (11) defined operators as symbols or combinations of symbols that affect

the value or ordering of an operand, and an operand is defined as being a variable or constant.

In a paper discussing Halstead's work, Elshoff (8:30) criticizes these definitions as not being specific enough and states that questions still remain about counting of operators and operands. In another paper, Salt (21:59-60) echoes Elshoff's comments about ambiguity resulting from Halstead's definitions. Salt cites the counting of the IF ... THEN ... ELSE construct as an example. One researcher considered this construct to be a single operator, but a second researcher claimed that the IF ... THEN and the ELSE were two distinct operators. In yet another paper, Misek-Falkoff (18:86-88) offers another example that is not easily resolved by using Halstead's definitions. That example is

```
X = F1 (F2 (Y) ),
```

where F2 is an operator with respect to Y and F1 is an operator with respect to F2(Y). It is unclear here whether F2 should be counted as an operator, as an operand or both. As pointed out by Beser (2:51), every experiment involving Halstead's work seems to use a counting strategy which is unique to that experiment. This difference in counting rules used by various researchers make comparison of their empirical results a difficult job.

Several experiments have been conducted to determine what impact, if any, different counting strategies have on the

software science metrics. Elshoff (8:30,40) counted a collection of 34 PL/1 programs using 8 different counting methods and found that the effects of the various counting methods varied depending on the characteristics being measured. Some of the metrics such as length, N, and volume, V, changed very little while level, L, and effort, E, varied significantly. He concluded that although no one counting scheme could be shown to be the best, the results did indicate the importance of the counting method to the overall measurement of an algorithm. In a separate study, Conte (5:118,126) modified Halstead's method of counting the GOTO construct. His results showed that this modified counting strategy had minor effects on N, Nhat, and V, but that it had significant impact on n_1, N_2, Lhat, and E.

In addition to the lack of consensus on how to count operators and operands, there is disagreement on what parts of a program should be included in this count. Halstead contended that declarative statements should not be included in the counting process and most research (13:59) has followed this lead. However, Kavi and Jackson (13:57,71) conducted an experiment with 'C' language programs in which declaration statements were included in the operator and operand count. They justified this departure from the normal practice by contending that declarative statements are an important part of an algorithm in most programming languages, and to a certain extent they determine the structure and complexity of programs. They state that this

line of thought is in accordance with the accepted belief that "Algorithms + Data Structures = Programs". From this point of view, the "algorithm" is the part of the program that is typically counted and the "data structure" is the declarative part that typically is not counted.

Salt (21:60) seems to convey the contemporary view on counting strategies when he says:

> There is clearly a need for more information about counting strategies in research papers. Certain aspects of the strategies require special attention. Although short descriptions of operands are acceptable, the same cannot be said about operators. Comprehensive descriptions of operators are required. General statements to the effect that operators are comprised of reserved words and special symbols are inadequate. Such statements leave too many unanswered questions. In PASCAL for example, is the reserved word NIL an operator? Particular attention is also required in the consideration of symbols with more than one function. For example, in FORTRAN, a set of parentheses may be used to delimit expressions, arguments, or subscripts. A counting strategy must be clear about how many unique operators are involved.

At this point, the presentation of Halstead's theory of Software Science ends and a review of the published findings begins.

Acceptance/Criticisms of Halstead's Software Science Theory

To become an effective tool for software engineers, Halstead's theory on Software Science must accurately predict information about a software project before the coding stage. Halstead (11:51-53) investigated the predictive power of his formulas by asking a computer scientist, who was fluent in three languages (FORTRAN, PL/1,

and APL), to program, in each of the three languages, 12 algorithms from the Communications of the Association for Computing Machinery. Using the software science equation in estimating programming time, Halstead predicted the time for the programmer to finish the job. The relationship between actual vs predicted programming time was very strong - a correlation of 0.94. The actual programming time was 14.68 hours, which compared well with the predicted time of 15.45 hours.

In another experiment, R. D. Gordon (10:11-13) measured the number of minutes needed to implement a program fully; this included the time to read the problem statement, to the finished product with no errors. The predicted time was within 3 percent of the actual total time with a correlation coefficient of 0.934.

Research conducted by the Computer Center of Purdue (11:14) observed that Halstead's work can predict the length of programming time, number of programming errors, and the quality of the final programs. Other independent statistical studies conducted by Kerlinger (10:10), Campbell and Standley (10:10), and Elshoff (11:14-16) have tested Halstead's formulas with impressive results, thereby enhancing the validity of his works.

A. Fitzsimmons and T. Love (10:10-17) discovered a pattern in all the experiments they reviewed concerning software science. This pattern seemed to indicate that there is a correlation of the effort measure with many

factors that affect programming projects such as programming time. Software science may be a possible tool to answer the questions considering the difficulties of programming and the causes of high software cost.

Although early studies have shown impressive results, software science has not been universally accepted (23:157-164) and is not being widely used outside the academic arena. Some have questioned the validity of the experimental data. In most cases, the sample size and programs were small. The experiments did not involve professional programmers, but a few college students who may not represent the typical programmer. The assumption that the human brain is capable of making a constant number (S) of mental discriminations per second is questionable (23:158; 6). Few psychologists today agree with the 'Stroud number' because of lack of empirical results.

As mentioned in section two, defining and counting operators and operands has been a major issue of concern because these tokens are the basic foundation of software science (23:157). The results of the experiments may depend on these definitions. For example, Halstead ignored the declaration section and other nonexecutable statements of the algorithm. Some have argued that nonexecutable code is a major part in determining programming time and must be counted. To make matters worse, classifying a token as an operator or operand may not be clear. The meaning may depend on the use of the token at execution time, i.e. a

function name may serve as both an operator and operand. Others have also suggested grouping operators, because different operators have different impacts. These ambiguities in interpretation of operators and operands may result in different values for some of the software science metrics (see Appendix A for an example of two counting schemes). Therefore, a standard counting strategy needs to be developed for languages in order to make valid and consistent decisions from the experiments; otherwise, experimental results will continue to vary and prove to be useless for project managers.

R. Wolverton noted (26:484-485) that Halstead's work is too advanced to be any practical use in estimating software production; however, if Halstead's theory is properly used and understood, it might be useful at some future time. One possible application is applying his theory to explain the compilation effort resulting in a performance metric which researchers could use to evaluate different compilers.

An Application of Software Science to Compilers

Although Halstead developed his model in an attempt to predict, among other things, the amount of time it will take for a computer programmer to write a given routine, it may also be useful to predict the time required for a compilation of that same routine. This section discusses, in general terms, the components of a compiler and the steps involved in the compilation process. Further, it

demonstrates that the compilation process and the process of a programmer writing a program are similar enough that it is reasonable to investigate the ability of Halstead's model to predict the time required to compile a given routine.

A compiler can be defined (24:5) as a translator which transforms a high-level language such as FORTRAN, PASCAL, or COBOL into a particular computer's machine or assembly language. A programmer can also be thought of as a translator because he transforms an English language problem statement into a high-level source language that can then be processed by the compiler.

A compiler has two major phases (24:6-11): analysis of the source program and synthesis of the object code for that program. Fig. 2.1 depicts this structure as well as the major sub-phases involved. This structure may vary between individual compilers and between compilers for different languages, but it is representative of a generalized compiler.

Analysis Phase. The major function of the lexical analyzer is to scan lines of the source program and separate the text into a sequence of tokens such as constants, variables names, reserved words, operators, and punctuation. This sequence of tokens is then passed to the syntactic analyzer which groups the tokens into larger syntactic classes such as expressions, statements, or terms. If the syntax analyzer determines that the token sequence is not

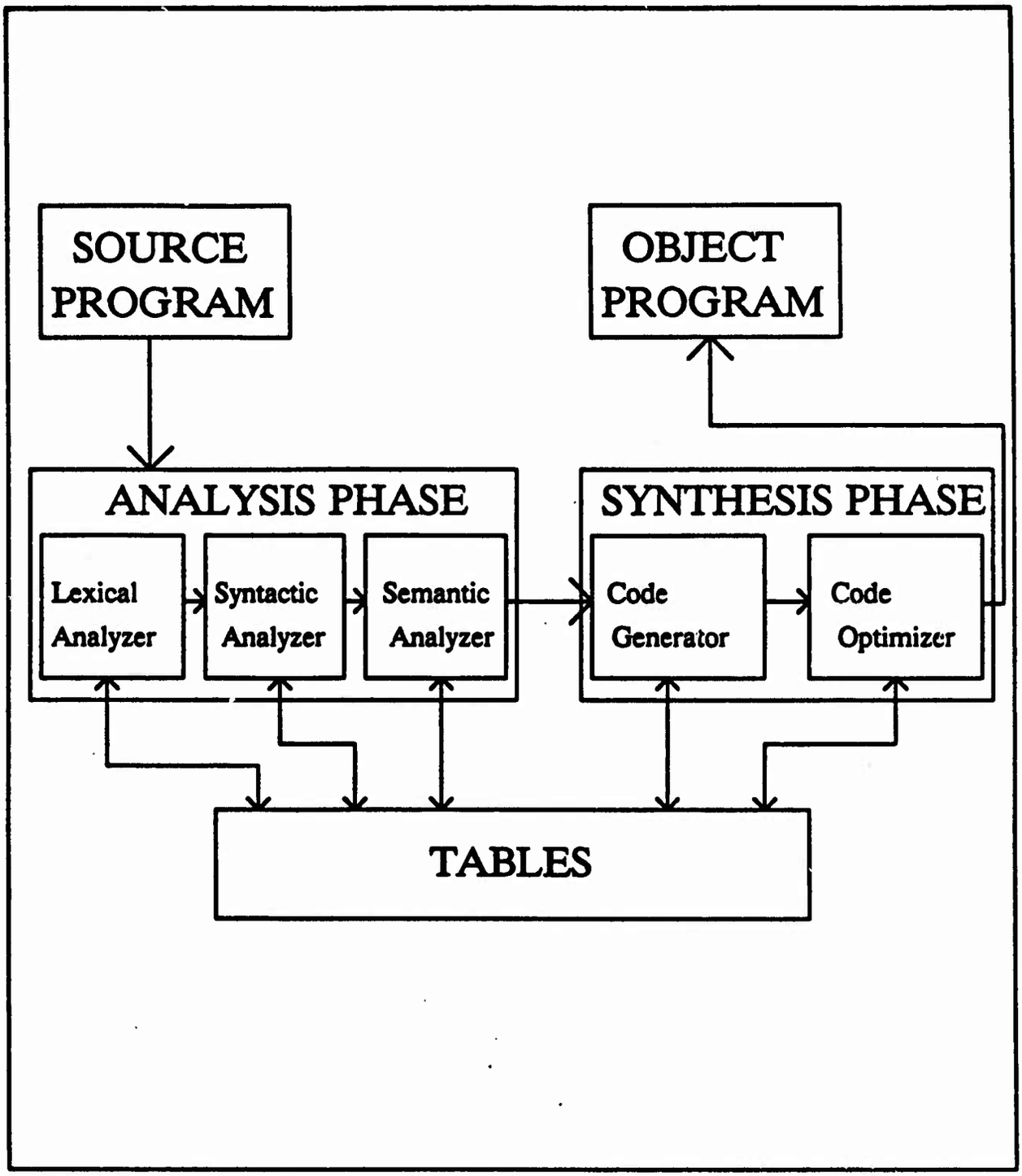


Fig. 2.1 Components of a Generalized Compiler. (24:6)

syntactically correct, it generates an error message. If the sequence is in the correct format, a syntax tree or equivalent structure is built for that sequence. The syntax tree is then passed to the semantic analyzer. The semantic

analyzer determines what actions are being requested. The semantic analyzer may produce some form of intermediate source code which will be passed to the synthesis phase of the compiler. Several structures, such as a symbol table, are built during the analysis phase of the compile process.

A programmer goes through similar steps in preparing to write a program, although in reality he probably performs them in parallel rather than serially as the compiler does. His lexical analysis probably will not break the problem statement down to the level of individual tokens, but he will break it down into paragraphs, sentences, and phrases to generate ideas and concepts about the structure of the problem that is to be solved by his program. The programmer's final step in the analysis phase is to perform a semantic analysis to understand exactly what the problem statement is asking for. As in the compiler process, tables and other structures may be built to aid in completion of the overall task. Logic diagrams, truth tables, and flowcharts are examples of these structures.

Synthesis Phase. The code generator, the first step of the synthesis phase, translates the data received from the analysis phase into either assembly language or machine language. In more sophisticated compilers, the output of the code generator is passed to a code optimizer where the code is evaluated to determine if it can be restructured to make it more time or space efficient.

A programmer also takes the results of his analysis phase and generates an output, the high-level language program. He may then analyze his program, much like a code optimizer would do, to see if some of it may be implemented more efficiently. In the case of the programmer, the search for efficiency is probably on-going during the entire synthesis phase.

Conclusion. There is not a one-to-one correspondence between all actions taken by the compiler and the programmer, but there are certain parallels. Both must input data, analyze that data to determine its validity and meaning, and determine what action that data is requesting. They both must generate a product, in a language different than that of the input data, that conveys the same information as the input data. Because the programming process and the compilation process have a number of similarities, it seems reasonable to expect that Halstead's model might predict compilation time.

Programming time may vary depending on the programmer's well being, state of mind and many other factors. As a result, the value of 'S' in Halstead's time equation, $T = V^2 / (S * V^2)$, is questionable since a programmer's discrimination rate or programming speed may vary day to day. In contrast, compilation time is solely based on the host computer and the program to be compiled. Therefore, the discrimination rate or, in this case, the translation

rate or processing speed of a compiler may be more deterministic. Consequently, an extension of Halstead's model may predict compile time even better than it predicts programming time.

Having presented the theory behind this thesis effort, the research methodology can now be presented.

III. Research Methodology

The extension of Software Science theory to a compiler is straight forward. Like a programmer, a compiler translates a language from one level to another. Consequently, it seems reasonable to apply a form of Halstead's programmer time equation to predicting the compilation time for a compiler. If a mathematical model can be developed, an important role for the compiler performance model would be to predict the compilation time for compilers. However, even more beneficial, would be the ability of the model to compare performance rates of various compilers.

This chapter describes the methodology involved in analyzing software science as a possible tool for explaining compiler time and for the development of a performance index. The first section presents the mathematical models investigated in this research effort. The second part describes the experimental design to validate the extension of software science to compilers.

Model Proposals

Software Science theory served as the basic theoretical framework for predicting compiler time. Three mathematical models for predicting compiler time will be presented; first, Model 1 based on the time equation; second, Model 2, a linear model based on program length; and finally, Model 3, a non-linear model based on program length. Model 1

utilized program volume and potential volume just as Halstead envisioned. Models 2 and 3 made use of Halstead's definition of length as defined in Chapter II. Although length is not part of Halstead's theory for predicting time, length is a common complexity measure used in estimating time to complete a task. Therefore, Models 2 and 3 were investigated for the purposes of comparing the predictive power of these models to Model 1. It is assumed for all models that a program to be compiled is syntactically correct and the program length, N, is greater than zero. All the models were analyzed to determine which, if any, were best suited for predicting compile time.

Model 1 - Using Software Science Time Equation. Model 1 used Halstead's programming time equation as the basic theoretical model. The equation was specified as a set of independent variables related by a set of parameters to be estimated. The dependent variable is the actual CPU time required for the compilation process. The volume, V and potential volume, V^* are the independent variables. Referring to the Time Equation 2.13,

$$T = V^* / (S * V^*),$$

and placing it in parameter form yields:

$$T = K * V^a * (V^*)^b. \qquad (3.1)$$

This equation is exactly the same as Halstead's time equation if 'K' is a fraction, 'a' is 2, and 'b' equals -1. 'K' has the same meaning in the compilation process as the constant 'S' in Halstead's equation for predicting programmer time. 'K' represents how fast the compiler does its job (the processing rate) or its discrimination rate. 'K' will depend on the computer architecture and the efficiency of the compiler itself. Clearly 'K' can be interpreted as a performance index given that 'a' and 'b' are known. Or, 'K' can be used in an estimation role to distinguish compilers.

Model 2 - Length: N (linear). It seems reasonable to assume that the more operators and operands in a program, the more time the compiler must expend on resources. This linear relationship can be shown as follows:

$$T = a * N, \qquad (3.2)$$

where 'T' represents compilation time and 'a' is some constant multiplier. As in Model 1, the dependent variable, 'T', is the actual CPU time required for the compilation process. However, in this case, the independent variable is the length, N.

Model 3 - Length: N (non-linear). Compile time may not behave in a linear fashion as a function of length. A common complexity measure for determining programming time is lines of code (LOC). "It is generally accepted that a program requiring more lines of code will take 'proportionally' longer to implement than another program requiring fewer lines" (21:160-161). It then seems logical that compiler time would behave in the same manner - the longer the program, the longer the compiler time. To relate the lines of code measure to actual programming time, a formula of the following type (21:160-161) can be derived using regression analysis:

$$T = a * LOC^{b}.$$

Using the same logic and replacing LOC with Halstead's definition of the length of a program, the model now becomes:

$$T = K * N^{b}. \qquad (3.3)$$

where T represents compiler time. Again, the dependent variable is the actual CPU time required to complete the compilation process. As in Model 2, N is the independent variable related by a set of parameters to be estimated.

Using Parameter Estimates. Halstead envisioned that obtaining the actual counts for some algorithms may be difficult or impractical. Therefore, Halstead defined estimators for certain parameters such as the length of an algorithm. Consequently, Halstead's measures can be divided into calculated and estimated equations. To determine the effect of these estimators, each model described above had two cases: one based on the calculated, and the other based on the estimated values. In Model 2 and 3, N was replaced with Nhat and was calculated using Equation 7 from Table 2.1. Model 1, Equation 3.1, replaced V and V^* with Vhat and Lhat from Table 2.1 where

$$1)\ \text{Vhat} = \text{Nhat} * \log_2(n),$$

$$\text{and } 2)\ \text{Lhat} = (2 * n_2) / (n_1 * N_2).$$

The Experiment Design

The experiment required a rich set of algorithms written in Ada. Next, the various software science measures described in Chapter II were calculated. This required the identification and enumeration of each operand, operator, and I/O variable in each program. The programs were then compiled on four computers using different compilers and the time to compile was recorded. The model equations were transformed to the linear models. Then using linear regression techniques in the SAS program package, the models were analyzed.

Data Selection. For the purpose of this investigation it was desirable to select a database that would guarantee that the results were statistically valid. Therefore, the desired approach was to use published or production software. The ACEC's programs seemed to be the perfect candidate for this study since DoD sponsored the creation of this benchmark test suite to validate and evaluate Ada compilers.

ACEC consists of a series of public domain test programs collected by the Ada E&V team for the Ada Joint Program Office. The programs provide information about language features that must be present in a compiler if it is a full implementation of the ANSI/MIL-STD 1815A. (12:3)

A copy of the ACEC test suite was obtained from SofTech, Inc., at the address below, who was contracted by the Air Force Wright Aeronautical Laboratories to distribute the programs.

SofTech, Inc.
Attn: Mr. Michael C. Hill
3100 Presidential Drive
Fairborn, OH 45324-2039

Approximately 300 modules currently exist in the test suite. The programs are divided in two categories called normative tests and optional tests. The normative tests (12:3) provide a means for determining system cost for a particular language feature, that is, collecting information on the

speed, space and the limitations of the Ada compiler. On the other hand, the optional tests (12:4) provide measurments of features that are not a required part of the Ada compiler.

Of the 300 test programs, 171 were selected for this investigation. Programs were eliminated if they included pragmas, or they were similar to other modules, i.e. the vocabulary and length were the same.

Identification/Enumeration of Operands, Operators, and I/O Parameters. Before any data could be analyzed using the software science metrics, a suitable set of rules for counting operators, operands, and I/O parameters had to be devised. In Halstead's original work, only executable operands and operators were counted because the theory was intended to analyze algorithms, not programs. However, a compiler must process all the tokens (operators/operands) in a program and can expend substantial resources translating data types, declarations, tasks, etc. Therefore, in this investigation, the counting strategy had to be expanded to include all tokens. Due to the importance of the operator and operand counting definitions, the counting strategy implemented is summarized in Table 3.1. See Appendix B for examples of counting Ada programs. For a detailed description of this strategy, refer to Captain Maness's 1986 thesis (17) on validating an extension of Halstead's theory

TABLE 3.1

ADA COUNTING STRATEGY

1. All entities in a module are considered, except comments.

2. Variables, constants, literals are counted as operands. Local variables with the same name in different procedures/functions are counted as unique operands. Global variables used in different procedures/functions are counted as multiple occurrences of the same operand.

3. The following pairs of tokens are counted as single operators:

And Then	Array Of	Begin End
Body Is	Case Is When End Case	
Declare Begin End		Do End
Elsif Then	Exception When	For In Loop End Loop
For Use	Function Return	If Then End If
Limited Private	Loop End Loop	Or Else
Record End Record	Select End Select	Subtype Is
While Loop End Loop		

4. The following tokens or pair of tokens are counted as single operators subject to the accompanying conditions:

+ is counted as either a unary + or binary + depending on its function. A unary + is not counted when it is a part of a numeric constant like +3.14.

- is counted as either a unary - or binary - depending on its function. A unary - is not counted when it is a part of a numeric constant like -2.15.

( ) is counted as either (1) an expression grouping operator, as in '(x+y)/z, (2) an invocation operator, as in xx := SQRT(a), (3) a declaration operator, as in Procedure xx (a:in real), (4) a subscript operator, as in x = I(j), (5) a dimensioning operator, as in k : array (1..6) of real, (6) an aggregate operator, as in x : f_type := (others => ' '), (7) an enumeration operator, as in type color is (red,green,blue), or (8) a conversion operator, as in int := integer(real_variable).

' (apostrophe) is counted as either (1) an attribute operator, or (2) an aggregate operator. A pair of apostrophes used in character constants, such as 'x' is counted as a single operator.

Table 3.1

ADA COUNTING STRATEGY (cont-)

in	is counted as either (1) a mode operator,or (2) a membership test operator.
or	is counted as either (1) a boolean operator, or (2) an alternative operator in select statements.
null	is counted as either (1) an operator if it appears in executable code, or (2) an operand when used as a constant.
private	is counted as either (1) a declaration operator, or (2) a detail operator.
separate	is counted as either (1) a declaration operator, or (2) a detail operator

5. The following tokens are counted as single operators if they are not used in rules 3 and 4:

*	/	**	&	,	.
..	:	;	/=	<	>
<>	=	=>	>=	<=	:=
|	<<>>	" "	# #	abort	abs
accept	access	all	and	at	constant
delay	delta	digits	else	end	entry
exception		exit	generic	goto	is
mod	new	not	out	others	package
procedure		raise	range	rem	renames
return	reverse	task	terminate		type
use	with	when	xor		

6. Procedure and function calls are counted as operators. Also nested function and procedure calls are counted as operators.

7. Type indicators are counted as either (1) an operand in its own declaration statement, or (2) an operator if it types a variable, function, or subtype.

8. 'Package/Procedure/Function Is New' is called a generic instantiation operator and is counted as one unique operator.

9. I/O Parameters are either (1) formal parameters within a subprogram specification, (2) function names, or (3) parameters that are passed globally and referenced within a module.

to Ada compilers. He includes the logic behind the development of this strategy, including how input and output parameters were counted.

Once the rules were established, the next step was obtaining the values of the software science parameters. Therefore, each program was counted manually to determine n_1, n_2, n_2^*, N_1, and N_2. The number of unique and total occurrences of tokens in each program were recorded on a data sheet (see Appendix C). Since manual counting is prone to error, the programs were counted twice. Capt Maness helped in the counting since he used the same data in his study. Appendix D summarizes the results of this effort.

Computer/Compiler Selection. Selection of a computer had to meet two criteria: 1) the computer had to be located on Wright-Patterson Air Force Base and be accessible for this research; and 2) a validated Ada compiler had to be available for the selected computer. As a result, four computers were selected:

1) The AFIT Academic Support Computer (ASC), a VAX 11/785 computer using the Verdix Ada compiler.

2) The AFIT Information Systems Laboratory (ISL), a VAX 11/780 computer using the Digital Equipment Corporation (DEC) Ada compiler.

3) The AFIT Classroom Support Computer (CSC), a VAX 11/785 computer using the DEC Ada compiler.

4) The ASD Information Systems and Technology Center, a Data General (DG) MV-8000-II computer using the ROLM/DG Ada Development Environment (ADE) compiler.

Procedures for gaining access to these resources are described in Appendix E.

Computer Environment. The ASC is a multi-user system located in the School of Systems and Logistics. It is a VAX 11/785 computer running the Berkeley 4.2 UNIX operating system. The hardware configuration consists of one 800/1600 bpi tape drive, three 456 megabyte disk drives, and one electro-static printer/plotter. The central processing unit is 32-bit with main memory consisting of 8 megabytes. Currently, version 5.1 of the Verdix Ada Compiler is installed on the system. The system supports 32 user terminals and 10 remote user terminals. Peak load occurs during the day from 0900 hrs to 1800 hrs with an average of 20 users. During 0200 hrs to 0600 hrs the load drops to an average of 2 users,

The DG MV/8000-II computer system is managed by the Language Control Branch, located in the ASD Information Systems and Technology Center (ASD/SI). The central system consists of a 32-bit central processing unit with 8 megabytes of real memory. Secondary storage devices include two 354 megabyte fixed disk drives and two 800/1600 bpi tape drives. Listings can be printed on a 600 lpm printer. The

system currently supports 24 user terminals operating under the AOS/VS version 5.6 operating system. Version 2.30 of the Ada Development Environment including the ANSI/MIL-STD-1815A version 2.30 Ada compiler is installed to support the development of the Ada programs. This computer is mainly used for programmer training and evaluating Ada programs. System usage varies from 0 users to 10 users. The computer is idle most of the time.

The ISL computer is housed in room 245 in building 640, the School of Engineering. This VAX 11/780 computer is a 32-bit machine with 8 megabytes of main memory. The hardware configuration includes a 1600 bpi tape drive, three 500 megabyte fixed disk drives, a 250 megabyte winchester drive, a x-y plotter, and a laser printer. The system currently supports 16 user terminals and one remote user terminal operating under VMS version 4.4 operating system. Version 1.2 of the DEC Ada compiler is installed on the system. Since this computer is mainly used for research, usage varies like the DG computer.

Finally, the CSC, located in the School of Systems and Logistics, is a VAX 11/785 computer running version 4.3 of VMS operating system supporting 32 user terminals and 10 remote user terminals. This system contains an 800/1600 bpi tape drive, a 600 lpm printer, two 456 megabyte disk drives, one 256 megabyte disk drive, and 8 megabytes of main memory. The DEC Ada Compiler, version 1.2, is installed on this system. Like the ASC, the CSC is a busy system supporting

faculty, students and AFIT staff personnel. System usage varies from an average peak of 20 users during the day down to an average of two users in the early mornings.

Compiler Time Measurements. A major issue in this study was measuring the time of the compilation process for a program as accurately as possible. On a multi-user system, compilation time cannot be measured simply by a stop watch because of the contention with other users. As a result, CPU time instead of wall clock time was used. The CPU times for the DG, ISL, and CSC were obtained by looking at the list or history file generated during the compilation process. Besides giving information about the compilation of a program, the file contained the amount of CPU time and wall clock time used to complete the compilation process. Although the ASC Ada compiler generated a similar file, the amount of time used was not given. Consequently, another method had to be devised. In this case, the UNIX system command 'time' was used which provided information on the total CPU time to complete a process.

Having each computer dedicated to this experiment would have been ideal. Since this was impossible, the programs were compiled three times each during a period when the number of users/processes on the computers were at the lowest. Therefore three experimental replications were completed. The results of this effort is summarized in Appendix F.

The ACEC benchmark test suite required the programs be compiled in a certain order as shown in Fig. 3.1. As indicated by this figure, the test routines required the programs IO_PACKAGE, CPU_TIME, and INSTRUMENT, repectively, to be compiled before the test routines. These modules are library packages used by the benchmark test modules. During the initial checkout to make sure the programs compiled on each computer, it was discovered that the time to compile would increase as the library size increased. As much as five seconds could be added to the CPU time if a program was compiled last instead of first. Therefore, to have the same environment for each benchmark test program,

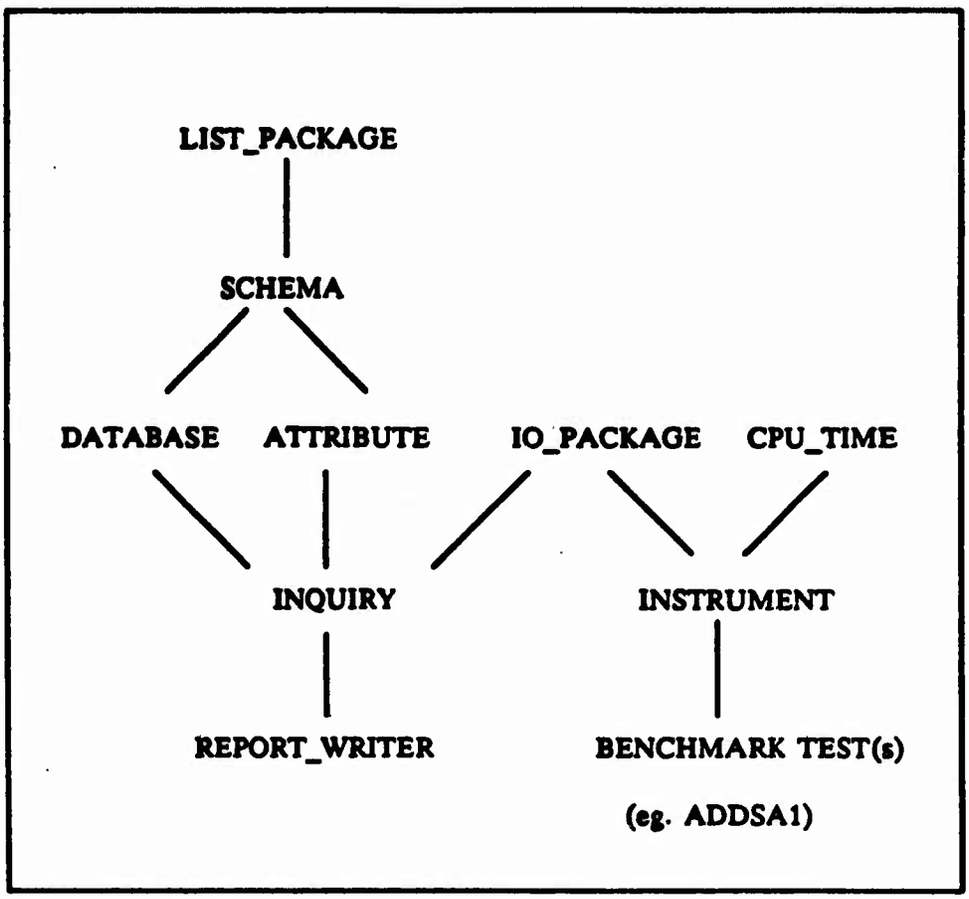


Fig. 3.1 ACEC Compilation Order (12:11)

the newly compiled test module was deleted from the library each time before the next compilation began. The programs were all compiled in batch mode. Basically, the batch job for each system consisted of the following:

1) Clean library directory - only the standard Ada library routines were presented at this time.

2) Compile in order IO_PACKAGE, CPU_TIME, and INSTRUMENT. (Note: compilation times for these programs were recorded)

3) Compile one benchmark test module such as ADDSA1, BALPA1, etc.

4) At the completion of the compilation process for the benchmark test module, delete all files generated, except the file containing the CPU times.

5) Repeat 3 - 4 until all programs are done.

6) Clean all files generated during the compilation except the standard Ada library routines.

7) Compile in order the following routines: LIST_ PACKAGE, SCHEMA, DATABASE, ATTRIBUTE, IO_PACKAGE, INQUIRY, and then REPORT_WRITER. (Note: These programs are not part of the benchmark test programs but provide the user the means of collecting statistics for the test suite. In this study SCHEMA, DATABASE, ATTRIBUTE, IO_PACKAGE, INQUIRY, and REPORT_WRITER are part of this database to be analyzed. LIST_PACKAGE was not included because it contained pragmas but it had to be compiled for use in the other programs.)

8) Repeat step 4.

All compilation times were than recorded, and the files containing the CPU times were deleted. Then steps 1 thru 8 were repeated two more times. Since each compiler has its own method of interacting with the user and the host operating system, the environment for each system is described below.

UNIX Environment. To be properly set up for running the Verdix Ada compiler on the ASC, the '.login' file must contain the path /usr/local/verdix5.1/bin. A test directory containing all the programs to be compiled was created. Each program needed a '.a' suffix for the compiler to recognize the source code to be compiled. The 'a.mklib' was used to make an Ada Library directory in the test directory. This utility created the necessary files and subdirectories where all files created and modified during the compilation process are placed. To compile all the programs in batch a shell script named 'compile' was created as follows:

```
clrall
time ada -v io_package.a
time ada -v cpu_time.a
time ada -v instrument.a
time ada -v addsa1.a
clr
time ada -v <program>.a
.
clr
.
time ada -v whlpa2.a
clrall
time ada -v list_package.a
.
time ada -v report_writer.a
```

To save time, two directories were created - one with just the standard Ada library routines, the other containing the standard Ada library routines and the library routines for IO_PACKGE, CPU_TIME, and INSTRUMENT. The 'clr_all' file (see Appendix G) is a script file which removes the file in the Ada library directories and moves a copy of a clean Ada library directory to the test directory. The 'clr' file (see Appendix G) also deletes the files in the Ada library directories, but moves a copy of the other directory after each benchmark test program is compiled in the test directory. The 'time' command records the time to complete the compilation process. The total time was determined by adding together the system and user time. The 'ada -v' commmand invokes the compiler and records a history of the compilation process.

To execute the shell script 'compile' in batch, the following command was entered:

```
compile >& acec.compile &
```

where acec.compile contains the history and times of the compilation process.

AOS/VS Environment. The DG ADE compiler interacts with the AOS/VS operating system through the Ada Development Environment. The Ada Development Environment was entered by typing 'enter'. Some preliminary steps were required before compiling a program. The first step was to create a project

directory by entering 'PROJCREATE'. This created the Ada directory where the source code (.ada suffix) was stored and the compilations were done. Next, the Ada library (.lib) file and the library searchlist (.lsl) file was created with the 'LIBCREATE' command. These commands were executed just once.

In this environment, all Ada compilations had to be done in BATCH. Therefore, a macro called 'compile.cli' was created to execute all the compilations in one job.

COMPILE.CLI MACRO:

```
ada IO_PACKAGE
ada CPU_TIME
ada INSTRUMENT
ada ADDSA1
clr
ada ADDSA2
.
.
clrall
ada LIST_PACKAGE
ada SCHEMA
.
.
ada REPORT_WRITER
clr
```

The compiler was invoked by entering 'ada'. The 'clr' and 'clrall' macros (see Appendix G) removed the newly compiled Ada programs except for the history file and returned the environment back to its original condition.

The compile macro was executed with the 'BATCH' command as follows:

```
BATCH compile
```

VMS Environment. As indicated before, both the ISL and CSC compilers interact with the VMS operating system. For these systems, the first step was to create an Ada program library directory where the compiler stores the files resulting from successful compilations. This was done by entering ACS CREATE LIBRARY [<MYDIRECTORY>.ADALIB]. Once this step was completed, it was not repeated. All newly compiled Ada programs, packages, and procedures are held here. Next, the current working library was defined by entering ACS SET LIBRARY [<MYDIRECTORY>.ADALIB] because a user may have multiple libraries in a directory. The compiler was invoked by entering ADA/<options> FILE_NAME. Each program to be compiled must have a '.ada' suffix.

To execute the compilation process in batch a macro called 'compile.com' was created. In this case, the macro consisted of the following:

```
     $ acs set library [mydirectory.adalib]
     $ ada/nooptimize/nocopy_source/nodebug/nonote_
source/lis cpu_time
     $ ada/nooptimize/nocopy_source/nodebug/nonote_
source/lis IO_PACKAGE
     $ ada/nooptimize/nocopy_source/nodebug/nonote_
source/lis INSTRUMENT
     $ ada/nooptimize/nocopy_source/nodebug/nonote_
source/lis ADDSA1
     $ acs del unit ADDSA1
       .
       .
     $ acs/nooptimize/nocopy_source/nodebug/nonote_
source/lis WHLPA2
     $ acs del unit WHLPA2
     $ acs del unit IO_PACKAGE
     $ acs del unit CPU_TIME
     $ acs del unit INSTRUMENT
     $ acs del unit ICS
     $ acs del unit HPSORT
```

```
     $ acs del unit X0*
     $ ada/nooptimize/nocopy_source/nodebug/nonote_
source/lis LIST_PACKAGE
        .
        .
     $ ada/nooptimize/nocopy_source/nodebug/nonote_
source/lis REPORT_WRITER
     $ acs del unit singly_linked_list
        .
        .
     $ acs del unit REPORT_WRITER
```

For this system, the compiler defaults to certain switches including optimization. Therefore, the above switches were set for the compilation including no optimization in order to have similar processing for each compiler. The 'acs del unit' command deletes the newly compiled program from the library.

Submitting a job in batch was accomplished by entering the following command:

```
submit/after=<date:time> compile
```

Statistical Analysis. The analysis of variance method and the linear regression tool on the SAS software package (22) was used to analyze the models and eventually answer the objective of this investigation. The data obtained in this study was used to test two hypotheses:

1) There is no significant difference in the predictive ability of Software Science in explaining compile time across alternative Ada compilers.

2) There is no significant difference in the discrimination rate across alternative Ada compilers.

Consequently, the analysis was divided into two parts. The first part investigated the model proposals and determined their explanatory power for various Ada compilers. Then Model 1's predictive power was analyzed. The second part investigated the development of a performance index to compare the speed of various Ada compilers.

Based on the set of data shown in Appendices D and F, the estimation of parameters was accomplished by specifying a regression equation with certain assumptions (18:408):

1) Error is a random variable that enters in the model in an additive fashion. The probability distribution of the error is normal with a mean of zero and a finite constant variance.

2) The error associated with one value of compile time has no effect on the errors associated with other compile time values, that is, the errors are independent.

The regression equation for Model 2 takes the form:

$$T = a * N.$$

Note that Model 1 and 3 are non-linear, and therefore had to be linearized for linear regression analysis. This can be done easily by taking the natural logarithms (Log) of both sides of the predicting equations. As a result, the regression equation for Model 1 now becomes:

$$\text{Log } T = \text{Log}(k) + a*\text{Log}(V) + b*\text{Log}(V^s),$$

and Model 3 yields:

$$\text{Log } T = \text{Log}(k) + b*\text{Log}(N).$$

Now the estimates of the unknown parameters for all the models can be calculated.

The data in Appendices D and F was first placed in a SAS data file as shown in Appendix H. Once this was done, a SAS command file containing the procedure to run regression analysis was generated/executed to analyze the above equations. A sample command file is contained in Appendix I. Then the various models' explanatory power, measured by the linear model coefficient of determination, were compared to each other. Next, Model 1's actual vs predicted compile times and correlation coefficients for each computer were investigated.

The next stage in analyzing the data involved determining if a performance index could be developed for the compile time prediction model (Model 1) based on Halstead's time

equation. For this analysis, dummy variables were used. A dummy variable is a simple way to observe the effect of each compiler and to analyze each compiler separately while maintaining the same exponents for volume and potential volume. In this case, two dummy variables (C,D) were needed to represent four compilers. As a result, Model 1's new regression equation becomes:

$$Log(T) = Log(K) + a*Log(V) + b*Log(V*) + e*C + f*D,$$

where e and f are the estimated values of the dummy variables that are added to the estimate of 'K'. The values of the dummy variables (C,D) were (0,0), (0,1), (1,0), and (1,1), where

(0,0) - Unix System

(0,1) - AOS/VS System

(1,0) - VMS-ISL System

(1,1) - VMS-CSC System.

Therefore, if e and f equals 2 and 3, respectively, then 0 would be added to the constant 'K' in the Unix system, 3 would be added to the AOS/VS system, 2 would be added to the VMS-ISL system, and finally, 5 would be added to 'K' in the VMS-CSC system.

The composition of the SAS data file from part 1 had to be changed for this analysis. Instead of 171 observations, the new data file contained 684 observations where the

compile times for each computer were aggregated in one column as shown in Appendix J. Also, two more columns were added for the dummy variables. Then, as before, a command file was generated/executed to determine if a performance index could be developed to compare the processing speed of various Ada compilers (see Appendix K).

Having presented the research methodology behind this investigation, the results can now be presented.

IV. Test Results and Discussion

As stated in Chapter III, the regression analysis tool in the SAS data analysis package was used in this investigation. This tool provided several statistical measures including the strength and the estimated parameters for a model. Appendix L gives a sample output from this tool and explains how a model is formulated using this output.

To demonstrate the feasibility of using an extension of Halstead's Software Science theory, it was necessary to show the explanatory power of using a mathematical model. Recalling the six ways of estimating the compile time:

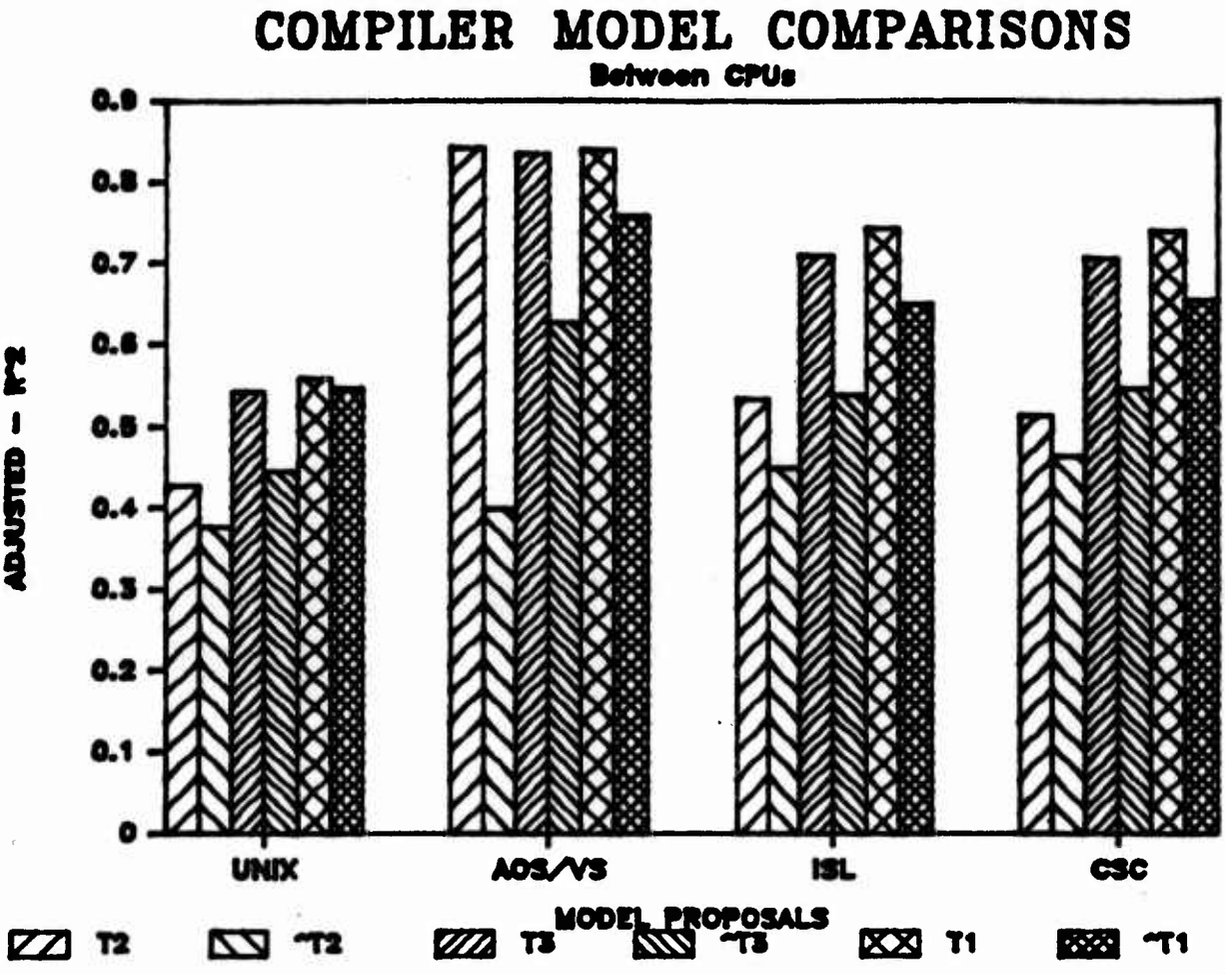
Table 4.1

Mathematical Models

Model	Calculated		Estimated	
T_1	$= K * (V^a) * (V^a)^b$	(1)	$K * (Vhat)^a * (Lhat)^b$	(4)
T_2	$= a * N$	(2)	$a * Nhat$	(5)
T_3	$= K * (N)^a$	(3)	$K * (Nhat)^a$	(6)

The adjusted coefficients of determination for each model were calculated and are depicted in Fig. 4.1. Although all the models were useful in explaining the compilation process of a compiler, overall, Model T_1 (Equation 1 in Table 4.1), using known V and V^a, provided the best explanatory power. That is, this model reduced the error the most in estimating

the compilation time for all the compilers over the average compile time. Therefore, the other models were abandoned, and Model T_1 was analyzed further.



Legend: $T1 = K*V^{a}*(V^{s})^{b}$ $\quad \hat{}T1 = K*(Vhat)^{a}*(Lhat)^{b}$

$T2 = A * N$ $\quad \hat{}T2 = a * Nhat$

$T3 = K * N^{a}$ $\quad \hat{}T3 = K * (Nhat)^{a}$

Fig 4.1 Compiler Model Comparisons

Table 4.2 shows the percentage of error reduction over the average compilation time if Model T_1 is used to predict compile time. It is interesting to note that the slowest compiler, AOS/VS system, provided the best model.

Table 4.2

Error Reduction in Predicting Compile Time

Computer	Mean Compile Time (CPU secs)	% Error Reduction
UNIX	13.36	55.56
AOS/VS	27.10	83.81
VMS-ISL	12.36	74.09
VMS-CSC	7.31	73.72

Based on the statistical analysis of T_1, the estimated exponents for V and V^s, 'a' and 'b' respectively are shown in Table 4.3. In Halstead's original work, he set the exponent of V and V^s in the programmer time Equation 2.13 to 2 and -1. As indicated in Table 4.3, the estimated exponents are approximately 0.5 and 0.1. For predicting compilation time, V^s does not appear to be as significant in the overall model as compared to Halstead's time equation. Instead of dividing V and V^s, the compiler model multiplied these two parameters, where V^s was very small. On the other hand, taking the square root of V is interesting because of the effect on modularization.

Table 4.3

Parameter Estimates

UNIX:	Adjusted R^2 = 0.5556,		F = 107.278	(0.0001)
	Parameter	Est	Std Err	Prob > T
	K	-0.6386	0.2075	0.0024
	a	0.4124	0.0315	0.0001
	b	0.0510	0.0292	0.0823
AOS/VS:	Adjusted R^2 = 0.8381,		F = 441.148	(0.0001)
	Parameter	Est	Std Err	Prob > T
	K	-1.5067	0.1415	0.0001
	a	0.5830	0.0215	0.0001
	b	0.0431	0.0200	0.0319
VMS-ISL:	Adjusted R^2 = 0.7409,		F = 244.079	(0.0001)
	Parameter	Est	Std Err	Prob > T
	K	-1.3314	0.1655	0.0001
	a	0.4730	0.0251	0.0001
	b	0.1047	0.0233	0.0001
VMS-CSC:	Adjusted R^2 = 0.7372,		F = 239.44	(0.0001)
	Parameter	Est	Std Err	Prob > T
	K	-1.7833	0.1642	0.0001
	a	0.4670	0.0249	0.0001
	b	0.0991	0.0231	0.0001

A program can be modularized, thereby reducing the volume. This can be expressed as:

$$V = \sum_{i=1}^{n} v_i$$

If Halstead's time equation is a function of the power of V and that power is greater than 1 then:

$$V^2 \gg \sum_{i=1}^{n} (v^i)^2$$

As a result, modularization reduces programming time. However, the compiler model, seems to indicate the opposite since the exponent was fractional. That is, if compile time is a function of the power of the volume, then the total sum of all the modules' volumes is greater than the one module containing all the programs:

$$V^a \ll \sum_{i=1}^{n} (v^i)^a, \text{ where } a < 1.$$

Consequently, compile time increases if modularization is used. The time reduced by a programmer when modularizing software causes the compiler to suffer in performance. Intuitively this makes sense, because the compiler must expend more resources checking the library and symbol tables.

The explanatory power of using Halstead's exponents, compared to the compiler Model T_1, is depicted in Fig. 4.2. Regression analysis was used to determined the constant 'K'.

Note that in all cases except for the AOS/VS compiler, Model T_1 did better. Therefore, having the exponents set to 2 and -1 for V and V^s, respectively, was rejected.

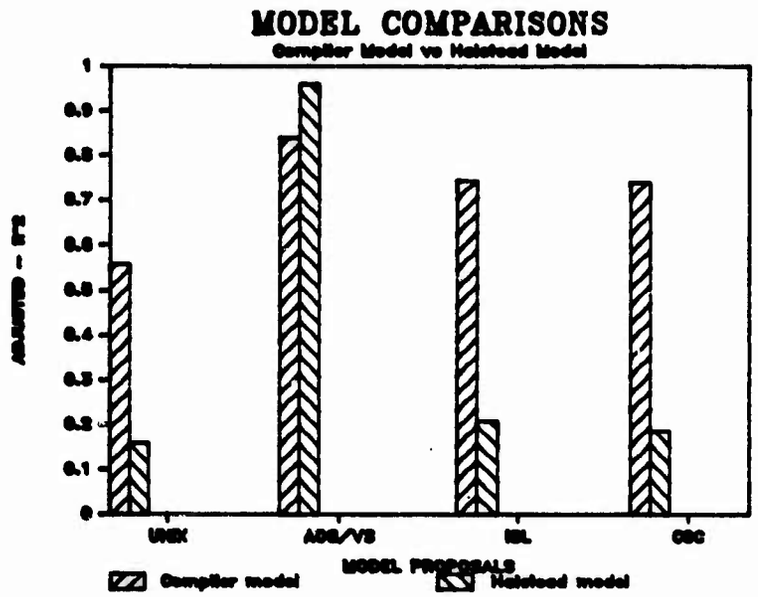


Fig. 4.2 Halstead Model vs Compiler Model

Based on Table 4.3, the estimated model for compilation time for each compiler is:

UNIX = T = $0.5281*(V^{0.4124})$

AOS/VS = T = $0.2216*(V^{0.5830})*((V^s)^{0.0431})$

VMS-ISL = T = $0.2641*(V^{0.4730})*((V^s)^{0.1047})$

VMS-CSC = T = $0.1681*(V^{0.4670})*((V^s)^{0.0991})$

If the independent variable was not significant - within .05, it was not included in the model. That is why, V^s is not in the UNIX model. Appendix M shows the actual versus the predicted times (using the above equations) for each module.

Table 4.4

Correlation Between Observed and Predicted (P) Compile Times

Variable	N	Mean	Std Dev	Sum	Min	Max
ASC	171	13.36	14.67	2285.17	3.23	93.73
DG	171	27.10	118.64	4633.31	5.08	1535.62
ISL	171	12.36	18.60	2113.06	2.79	126.40
CSC	171	7.31	10.63	1250.22	1.69	80.88
PASC	171	10.41	7.29	1779.45	3.68	63.93
PDG	171	18.57	22.91	3175.31	3.55	201.02
PISL	171	10.84	10.72	1853.80	2.63	72.59
PCSC	171	6.48	6.23	1108.60	1.62	41.82

where ASC - UNIX Compile Times
DG - AOS/VS Compile Times,
ISL - VMS-ISL Compile Times,
CSC - VMS-CSC Compile Times,
and the P prefix represents predicted compile times.

PEARSON CORR. COEFF. / PROB>/R/ / UNDER HO:RHO / N = 171

	ASC	DG	ISL	CSC	PASC	PDG	PISL	PCSC
ASC	1.0	.55	.86	.87	.74	.77	.79	.79
DG		1.0	.61	.59	.70	.75	.59	.59
ISL			1.0	.99	.80	.85	.88	.87
CSC				1.0	.80	.84	.88	.88
PASC					1.0	.98	.94	.94
PDG						1.0	.96	.97
PISL							1.0	.999
PCSC								1.0

Note: Significant Level - 0.0001

As demonstrated in Table 4.4, the correlation between predicted and observed compilation times for each compiler are all quite high. Consequently, the model fits well. Note also that the correlation between that actual times on the VAX computers (ASC, ISL, CSC) are quite high. This indicates that if the compile time on one VAX computer is high, then the compile time on another VAX computer will be high. This makes sense, because these computers are from the same family.

The predicted times compare relatively well with the actual times as shown in Fig. 4.3 thru 4.6. In these figures, all observations were sorted on the UNIX compile times which explains why the times on the AOS/VS system are not as smooth as the UNIX system. It was necessary to leave the last two data points on Fig. 4.4 out to make the graph presentable. Note on the residual graphs, as the compile time increases, the difference between actual and predicted times increases. This seems to suggest that the error does not enter the model in an additive fashion but exponentially. However, below 30 seconds of CPU time the model does very well. As a final note, it is of interest to observe that the large discrepancies between actual and predicted compile times occurs in the same location for each compiler. The magnitude of the error varies however, see Table 4.5 for a few examples.

Table 4.5

Residual Error Comparison

PROGRAM	SYSTEM	RESIDUALS
INTDA2	UNIX	-11.59
INTDA2	AOS/VS	-16.93
INTDA2	VMS-ISL	- 5.71
INTDA2	VMS-CSC	- 4.32
INTDB2	UNIX	-19.29
INTDB2	AOS/VS	-33.71
INTDB2	VMS-ISL	- 4.85
INTDB2	VMS-CSC	- 2.16

Capt Maness (17) investigated this phenomenon on the UNIX system, but was unsuccessful in determining a pattern in the

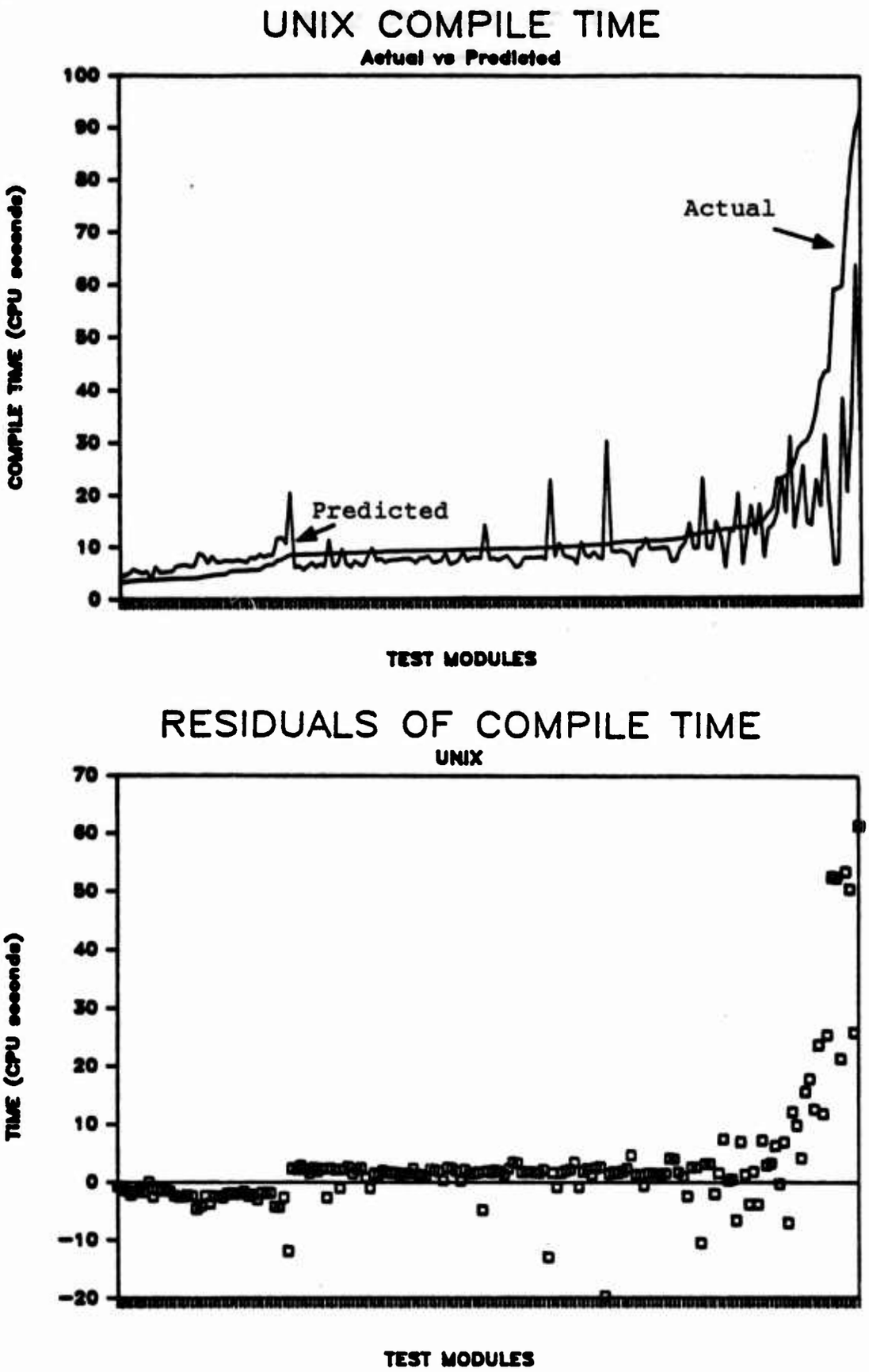


Fig. 4.3 UNIX Compile Time: Actual vs Model Prediction

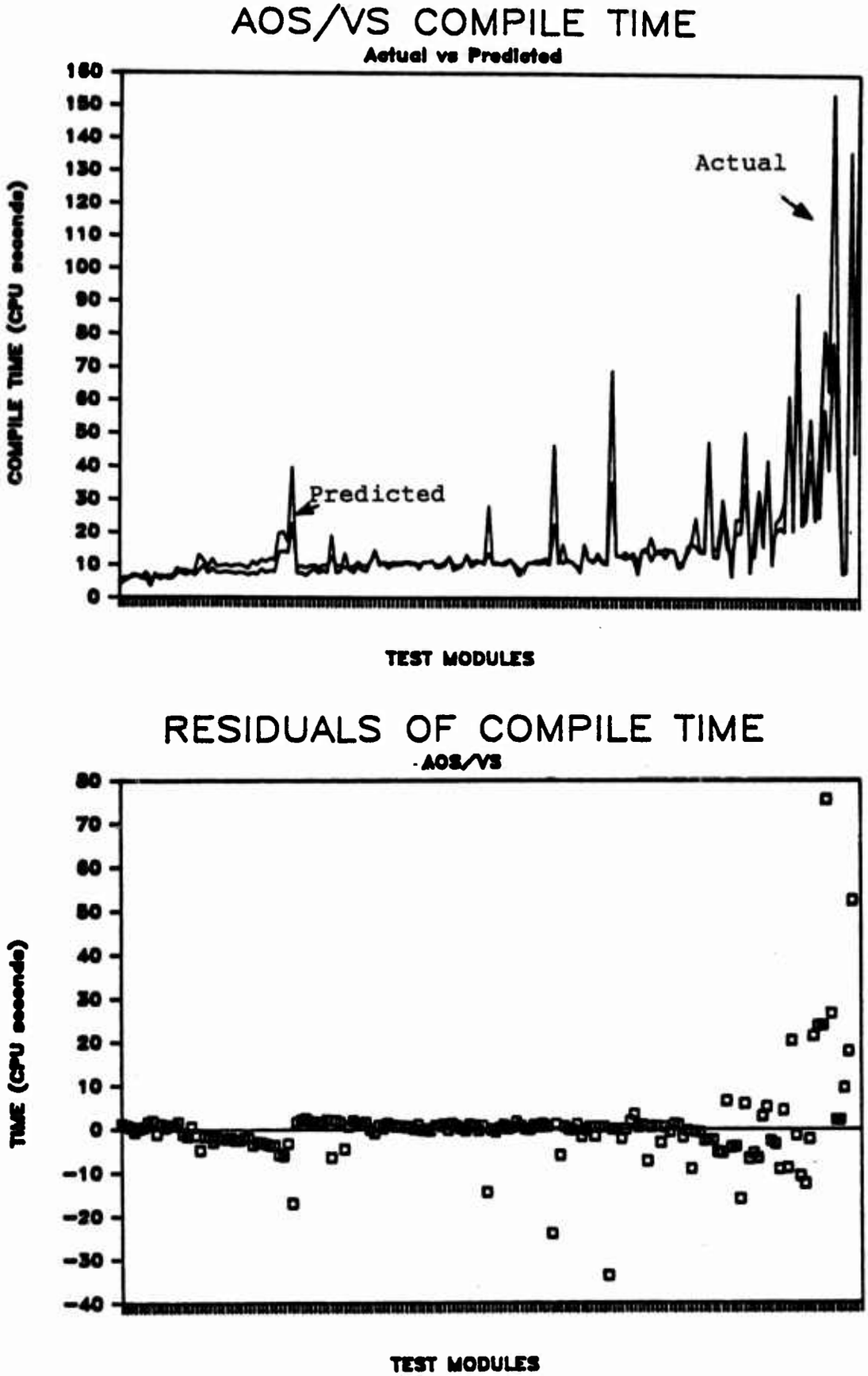


Fig. 4.4 AOS/VS Compile Time: Actual vs Model Prediction

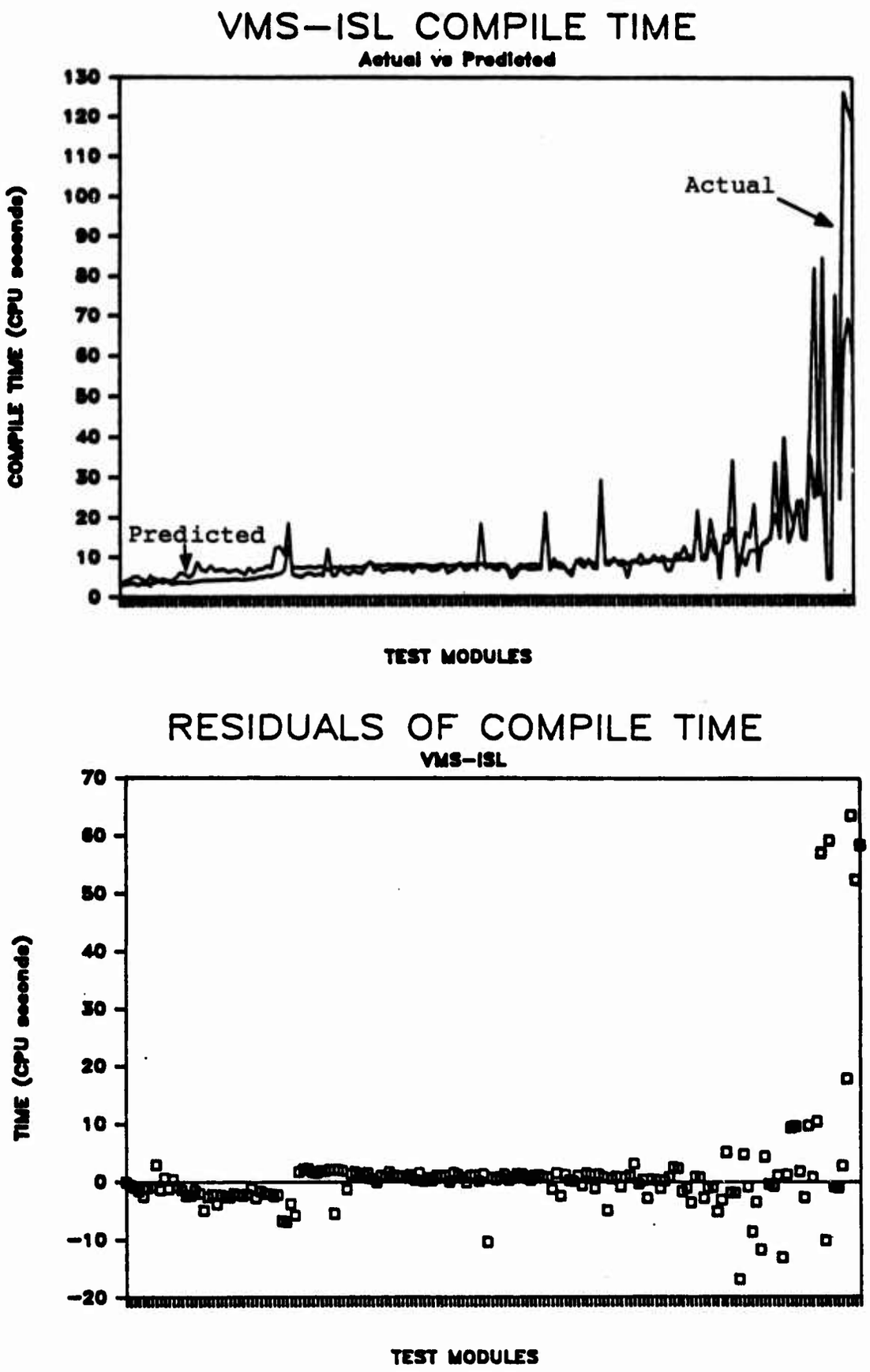


Fig. 4.5 VMS-ISL Compile Time: Actual vs Model Prediction

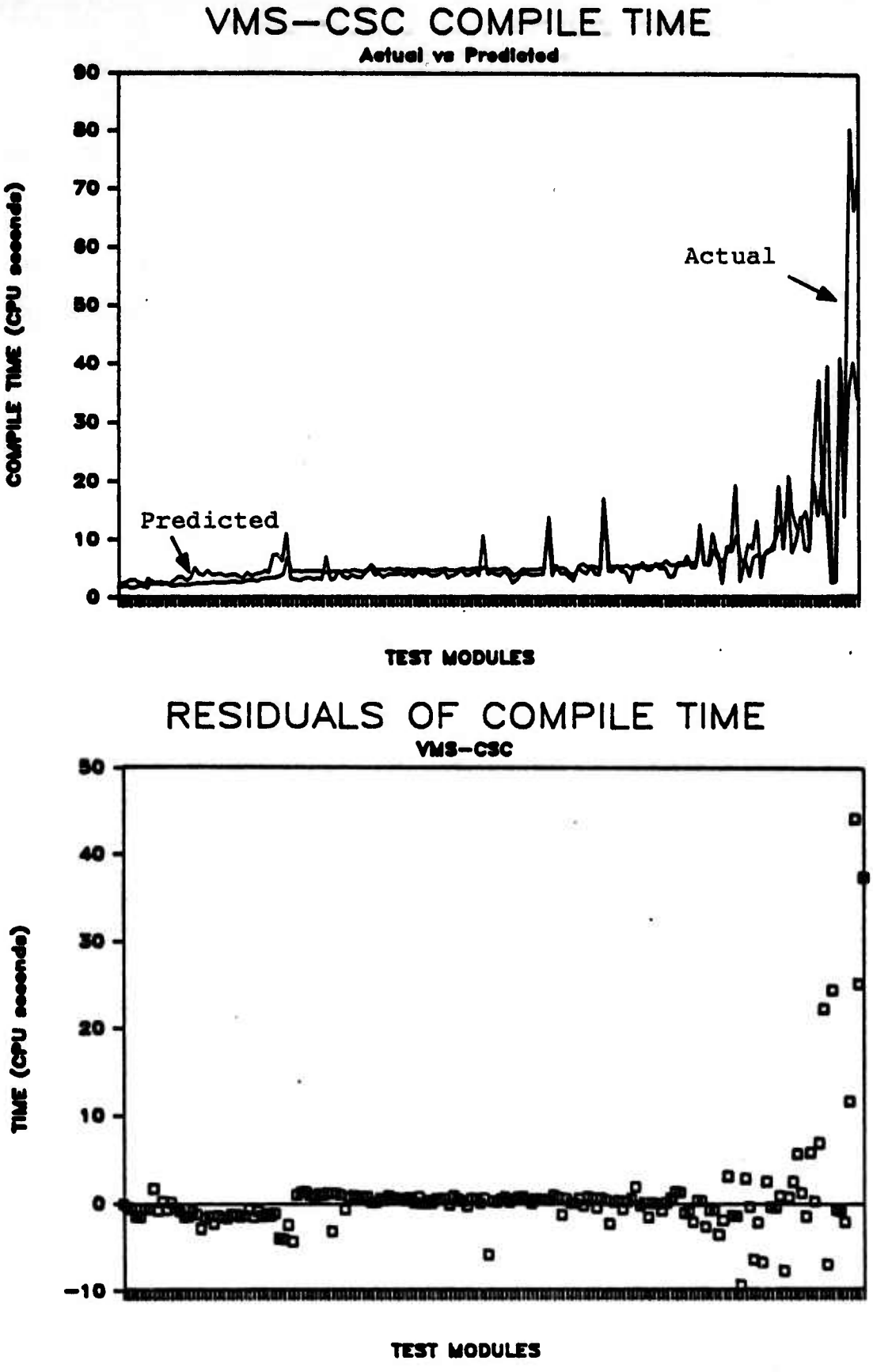


Fig. 4.6 VMS-CSC Compile Time: Actual vs Model Prediction

modules which would cause the large difference between the compilation times. Note that the predicted times are very close to the actual times, and then move apart as compile time increases past 30 seconds of CPU time.

Table 4.6

Parameter Estimates for Pooled Data

Adjusted R^2 = 0.7066, F = 412.312 (0.0001)

Parameter	Est	Std Err	Prob > T
K	-0.9818	0.1018	0.0001
a	0.4839	0.0151	0.0001
b	0.0745	0.0140	0.0001
Dummy e	-0.5601	0.0319	0.0001
Dummy f	-0.1063	0.0319	0.0009

The next major area of investigation was the development of a performance index. Table 4.6 shows the results of this effort. The results of using dummy variables does not really change the compiler model from Table 4.3. Note that the exponents for V and V^* are approximately the same - 0.5 and 0.1, respectively. However,'K', the translation rate, changes slightly. That is, the discrimination rate for each compiler is significantly different from the base, UNIX. The equations for each compiler now become:

$$\text{UNIX} = T = 0.3746*(V^{0.4839})*((V^*)^{0.0745})$$

$$\text{AOS/VS} = T = 0.3369*(V^{0.4839})*((V^*)^{0.0745})$$

$$\text{VMS-ISL} = T = 0.2140*(V^{0.4839})*((V^*)^{0.0745})$$

$$\text{VMS-CSC} = T = 0.1924*(V^{0.4839})*((V^*)^{0.0745})$$

Fi . 4.7 shows the performance rate of each compiler. Note th : each line represents the plot of the linear equation fo each compiler model. Appendix N shows a graph of the ac ıal model equations. As Fig 4.7 suggests, the compilers wo ld be ranked as follows (from the slowest to the fa :est):

Table 4.7

Ada Compiler Evaluation

Rank by Compile Ave.	Rank by Compiler Model	% Faster than UNIX
AOS/VS	UNIX	
UNIX	AOS/VS	10.1
VMS-ISL	VMS-ISL	42.9
VMS-CSC	VMS-CSC	48.6

Observe also that the compiler on the ISL and the CSC co puters were the same. Therefore, it can be concluded fr ı above that the CSC computer is faster than the ISL co puter by 10.1 percent. This is reasonable, since the CSC VA 11/785 computer is an upgrade from the ISL VAX 11/780 co puter.

Having presented the results of this research effort, a nu ber of conclusions and recommendations can now be pr sented.

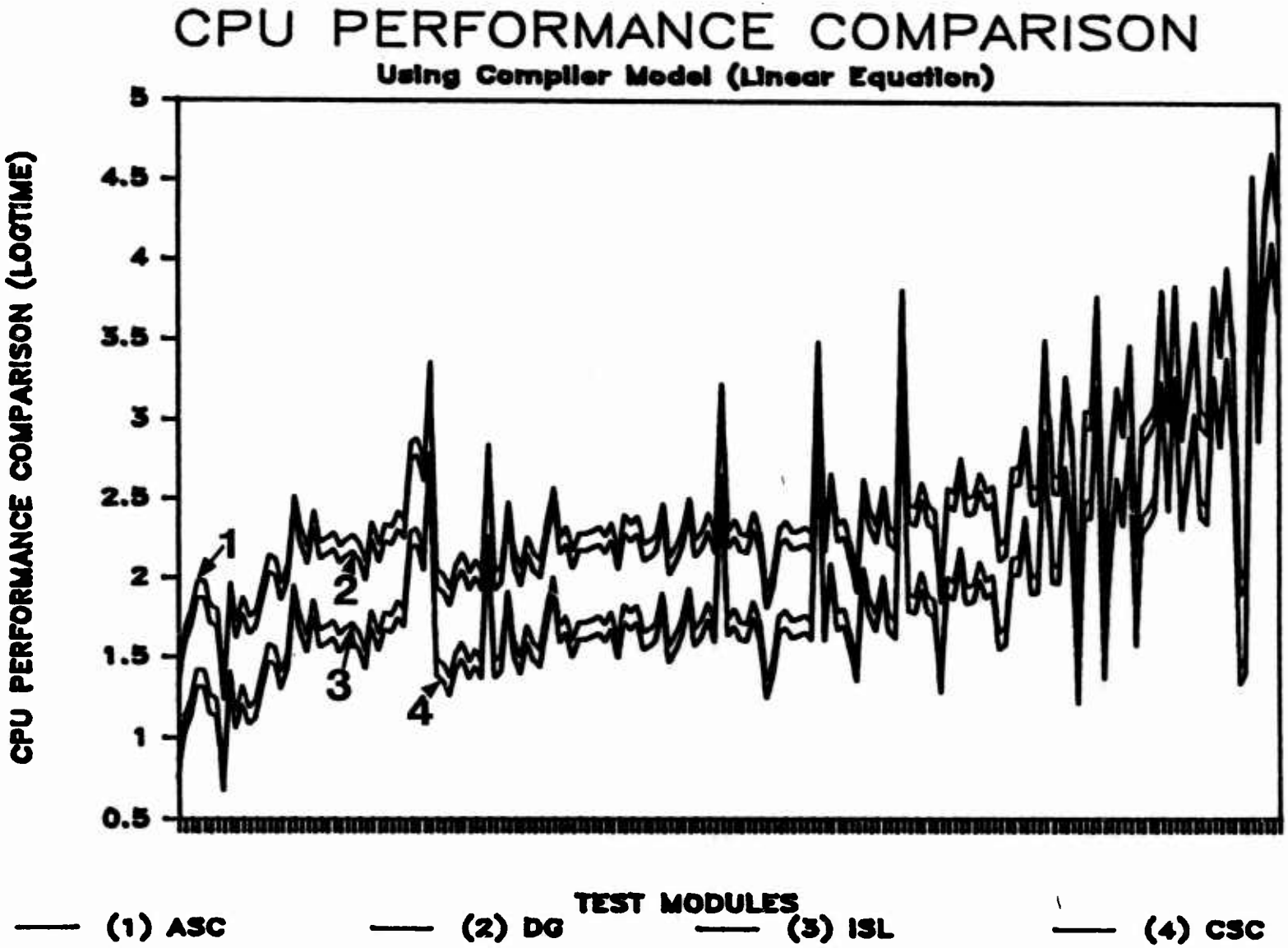


Fig. 4.7 CPU PERFORMANCE COMPARISON

V. Conclusions and Recommendations

In this study, an application of Halstead's Software Science theory to compilers was evaluated. Two basic concepts and properties of software science were reviewed, program length and programming time. Counting rules were established, and then software science measures examined the data collected. In addition, mathematical models were proposed and the appliciabilty of Halstead's theory to compilers was demonstrated. The experiment was designed to test two hypotheses used to validate this application. Recalling the null hypotheses -

1) There is no significant difference in the predictive ability of Software Science in explaining compile time across alternative Ada compilers,

2) There is no significant difference in the discrimination rate across alternative Ada compilers.

Evidence was presented in Chapter IV which supported the rejection of these hypotheses and, consequently, accepting the alternative hypotheses. That is, there is a difference in predicting compilation time for alternative compilers and there is a difference in the translation rate for different compilers.

Conclusions

A number of conclusions may be drawn from the above analysis:

First, the attempt to develop a measure which would provide a suitable approximation of the amount of time expended during the compilation process has been validated. The results suggest that the software science compiler model is a good tool for predicting compilation times. The correlation between the actual and predicted compile times were quite high:

Table 5.1

Correlation between Actual and Predicted Compilation Times

SYSTEM	CORRELATION
UNIX	0.74
AOS/VS	0.75
VMS-ISL	0.88
VMS-CSC	0.89

Second, using the actual value of the variables in the model provided a better approximation of compilation time. Referring to the model equations, all the estimated models used the length estimator. It as been shown (28:706) that the length estimator over-estimates small programs and under-estimates large programs. This is especially true with a powerful language like Ada that has a variety of operators available. Consequently, the compiler models using estimated values are not as accurate.

Third, the signs and the magnitudes of the estimated parameters were not within the proximity of the theorized values. In particular, the value of the exponent for the volume (V being approximately 0.5 instead of 2.0) was unexpected. However, this value seems reasonable. As explained in Chapter IV, a compiler must expend more resources compiling several modules separately than compiling a single program containing all the modules. In addition, potential volume, V^* was not negative and was not as significant in this application as compared to programming time. If the exponents were restricted to those proposed by Halstead's time equation, the explanatory power of the model was reduced as shown in Figure 4.2.

Fourth, the explanatory power of the model differed for each compiler. Therefore, the predictive ability of software science in explaining compile time across alternative Ada compilers was different, but uniformly encouraging. There are several possible explainations for this. First, different architectures and operating environments will vary the compile time for each compiler. Consequently, the estimated parameters may be inaccurate. Ideally, a dedicated system would have been preferred. Second, the efficiency of a compiler affects the compilation time.

Finally, one of the most significant findings in this study was the development of a compiler performance index. Clearly, from the results, the values for 'K' represents the processing rate of a compiler. The results indicate that

ranking compilers performance solely on average compile time is incorrect. Based on the average compile time, the ranking from the slowest to the fastest would have been the AOS/VS compiler, the UNIX compiler, and then the VAX compiler. In contrast, the software science compiler model ranking was UNIX, AOS/VS, and then VMS. As the results have shown, although the compiler on the AOS/VS system appears to be the slowest, it was faster than the compiler on the UNIX system as far as translation or processing rate. It was also shown that 'K' can also rank computers, if the operating environment is the same. In this case, as expected, a VAX 11/785 was faster than a VAX 11/780.

Recommendations

The following are recommendations for research on topics related to this thesis:

- Different Counting Rules. Defining and counting operators and operands is a major concern because these tokens are the basic foundation of software science. Therefore, different counting strategies should be investigated to determine their affect on the compiler model.

- Data Selection. By selecting a wider class of programs, the ability of the proposed model to account for various aspects of the compiler phenomena may be assessed.

- Nonlinear Analysis. As the results have indicated, the model becomes less effective as the compile time increases. Therefore, the assumption that the error enters the model in an additive fashion might be incorrect. Consequently, this error should be investigated further. In addition, a nonlinear model should be analyzed which assumes that the error enters the model in a multiplicative fashion.

- The Use of Pragmas. Unlike other languages, Ada provides pragmas which are directives to the compiler. The effects of using this construct on the compiler model should be investigated.

- Performance Index. This study could not determine which compiler was better because each compiler was on a different computer with different operating systems. The analysis only suggested that a compiler on a certain computer performed in a certain fashion. Consequently, if a user had to select a computer and compiler, this approach is appropriate. However, the only true test to determine which compiler is more efficient on a particular computer is to have all compilers in question on the same computer. Therefore, if possible, this should be investigated.

- Compilers for Other Languages. It is apparent that the software science compiler model is useful in predicting the compile time and determining the efficiency of a compiler.

Since only a limited analysis on a single language was performed, other languages should be studied.

This study obviously represents a preliminary exploration of the applicability of software science metrics to compilers. The results have indicated that there is enough evidence to continue investigating this area. Future research testing this model on other compilers and on a broader spectrum of data might illuminate the compilation phenomena of compilers. A compiler index has been proposed, developed and tested. With further research, the software science compiler model may become a valuable tool in evaluating compiler efficiency for the DoD and the civilian community.

APPENDIX A

FORTRAN Counting Rule Comparisons (2:65)

FORTRAN-IV element	PURDUE COUNTING RULE	SAP/H RULE
ACCEPT	Counted	Not Counted
BACKSPACE	Counted	Not Counted
CALL	Counted paired with routine name	Counted same as Purdue rule
DATA	Counted	Not Counted
DO	Counted	Counted, Paired with =,,
END	Not Counted	Not Counted
ENDFILE	Counted	Not Counted
GOTO Label	Counted As GOTO Label	Counted as GOTO Label is operand
GOTO(),VAR	Counted	Counted
IF()STATEMENT	Counted, () separate	Counted,grouped with ()
IF()LABEL,LABEL, LABEL	Counted, each label is separate GOTO labels	Counted as IF() Labels not counted
PRINT	Counted	Not Counted
READ	Counted	Not Counted
RETURN	Counted	Not Counted
REWIND	Counted	Not Counted
STOP	Counted	Not Counted
TYPE	Counted	Not Counted
WRITE	Counted	Not Counted
Var=Expression	Evaluated and counted	Evaluated and counted

FORTRAN Counting Rule Comparisons (Con't)

FORTRAN-IV element	PURDUE COUNTING RULE	SAP/H RULE
=	Counted	Counted, except from DO
Comma (,)	Counted, when in counted statement	Counted, when in counted statement
()	Counted, when in counted statement	Counted, from arithmetic express.
+ - * / **	Counted	Counted
Logical Operators	Counted	Counted
END OF STATEMENTS	Counted	Counted
Function Calls	Counted as operators and operands	Counted as operators when used in arith statements, else as operands
'LITERAL STRINGS'	Counted as operands	Not Counted
Subscripts	Counted	Not Counted
Variables	Counted as operands	Counted as operands if in counted statements.
I/O Variables	Counted	Not Counted

APPENDIX B

Ada Programs to Demonstrate Counting

EXAMPLE 1 - NRPCA1 Source Listing:

```
with INSTRUMENT;
use INSTRUMENT;

procedure NRPCA1 is
package PS_CS is new PROCS(INTEGER);
use PS_CS;
package B is new PROCS(BOOLEAN);
  TTRUE : B.T := B.T(TRUE);
  TFALSE : B.T := B.T(FALSE);
  TEST : B.T := B.Ident(TFALSE);
  Recursion : B.T := B.Ident(test);
  procedure Nested_Recursive_Procedure is

    Local_1 : T;
    Local_2 : T;

    procedure Nested is
    begin
      if BOOLEAN(Recursion) then
        B.Let(Recursion, B.Ident(TFALSE));
        Nested_Recursive_Procedure;
      else
        B.Let(Test, B.Ident(TFALSE));
        if BOOLEAN(Test) then
          Nested_Recursive_Procedure;
        end if;
      end if;
    end Nested;

begin
  if BOOLEAN(Recursion) then
    Let(Local_1,Ident(Init));
    Nested;
  elsif not BOOLEAN(Test) then
    Let(Local_2,Ident(Init));
    Nested;
  end if;
end Nested_Recursive_Procedure;

begin
START("NRPCA1","Nested Recursive Procedure Call (Control)");
for I in 1..100000 loop
  b.let(recursion,B.ident(test));
  Nested_Recursive_Procedure;
end loop;
STOP;

end NRPCA1;
```

The COUNT for NRPCA1:

OPERATOR	COUNT
NOT	1
BEGIN END	1
ELSE	1
ELSIF THEN	1
FOR IN LOOP END LOOP	1
IF THEN END IF	3
IS	3
GENERIC INSTANTIATION (NEW)	2
PROCEDURE	3
USE	2
WITH	1
" "	2
..	1
:	6
:=	4
;	30
,	6
( ) (aggregate)	2
( ) (invocation)	13
( ) (type conversion)	6
.	14

OPERAND	COUNT
INSTRUMENT	2
NRPCA1 (2 types)	3
PS_CS	2
PROCS	2
INTEGER	1
BOOLEAN	1
B	15
TTRUE	1
TFALSE	4
T	2
TRUE	1
FALSE	1
TEST	6
RECURSION	5
NESTED_RECURSIVE_PROCEDURE	2
LOCAL_1	2
LOCAL_2	2
NESTED	2
INIT	2
NESTED RECURSIVE PROCEDURE CALL (CONTROL)	1
BOOLEAN	4
I	1
1	1
100000	1

**the following operators
**are procedure calls

LET (2 types)	
first type	3
second type	2
NESTED_RECURSIVE_PROCEDURE	2
NESTED	2
START	1
STOP	1

**the following operators
**are function calls

IDENT (2 types)	
first type	5
second type	2

**the following operators
**are type indicators

T	6

EXAMPLE 2 - OPCEA1 Source Listing:

```
with INSTRUMENT;
use INSTRUMENT;

procedure OPCEA1 is

package NEW_PROCS is new PROCS(INTEGER);
use NEW_PROCS;
  Global_1: T;
  Global_2: T;
  Global_3: T;
  Global_4: T;

  function Function_1 (Input : T) return T is
  begin
    if Input = Init then
      return Init/Init;
    end if;
    return Function_1(Init);
  end Function_1;

  function Function_2 (Input : T) return T is
  begin
    if Input /= Init then
      return function_2(Init);
    end if;
    return Init/Init;
  end Function_2;

begin
  START("OPCEA1","Optimization Perf., Call Elim. (control)");
  for I in 1..1000 loop

  Let(Global_1, Ident(Init));
  Let(Global_2, Ident(Init));
  Let(Global_3, Ident(Init));
  Let(Global_4, Ident(Init));

  if Ident(Init) = Init then

Global_1 := T(T(Function_2(Init)*Global_4)/Function_1(Init);
  else
Global_2 := T(T(Function_1(Init)*Global_4)/Function_1(Init);
  end if;
  Let(Global_1, Ident(Init));
  Let(Global_2, Ident(Init));
  Let(Global_3, Ident(Init));
  Let(Global_4, Ident(Init));
  end loop;
  STOP;

end OPCEA1;
```

The COUNT for OPCEA1:

OPERATOR	COUNT
=	2
/=	1
*	2
/	4
BEGIN END	3
ELSE	1
FOR IN LOOP END LOOP	1
FUNCTION RETURN	2
IF THEN END IF	3
IS	3
GENERIC INSTANTIATION (NEW)	1
PROCEDURE	1
USE	2
WITH	1
RETURN	4
" "	2
..	1
:	6
:=	2
;	31
,	9
( ) (aggregate)	1
( ) (declaration)	2
( ) (invocation)	24
( ) (subscript)	4
**the following operators	
**are procedure calls	
START	1
STOP	1
LET	8
**the following operators	
**are function calls	
FUNCTION_1	4
FUNCTION_2	2
IDENT	9
**the following operators	
**are type indicators	
T	8

OPERAND	COUNT
INSTRUMENT	2
OPCEA1 (2 types)	3
NEW_PROCS	2
PROCS	1
INTEGER	1
GLOBAL_1	4
GLOBAL_2	4
GLOBAL_3	4
GLOBAL_4	4
FUNCTION_1	2
FUNCTION_2	2
INPUT (2 types, 2 each)	4
INIT	22
I	1
1	1
1000	1
T	4
OPTIMIZATION PERF., CALL ELIM. (CONTROL)	1

APPENDIX C

SAMPLE DATA SHEET

PROGRAM NAME:

Operators	Freq	Operands	Freq	I/O
(Logical)				
and				
or				
xor				
(Relational)				
=				
/=				
<				
>				
.				
ETC				
.				
(Binary)				
+				
-				
(Unary)				
+				
-				
(Multiply)				
*				
/				
mod				
rem				
(Highest Prec)				
**				
abs				
not				
(Short Circuit)				
and then				
or else				
(Char Entities)				
" "				
# #				
..				
:				
:=				

```
  <<>>            :          :                :          :
() expression     :          :                :          :
() invocation     :          :                :          :
   .
  ETC
   .

(Reserved Words)
  abort           :          :                :          :
  accept          :          :                :          :
  access          :          :                :          :
  all             :          :                :          :
  array of        :          :                :          :
  at              :          :                :          :
  begin end       :          :                :          :
  body is         :          :                :          :
  case is when
   end case       :          :                :          :
      .
      .
    ETC
      .
      .

PROCEDURE CALLS   :          :                :          :



FUNCTIONS CALLS   :          :                :          :



TASK CALLS        :          :                :          :



TYPE INDICATORS   :          :                :          :
```

APPENDIX D

Data for the Compile Time Study

Program	n_2*	n_1	n_2	N_2	N_1
ADDSA1	0	19	14	20	36
ADDSA2	0	19	14	32	48
AKERA2	3	24	12	28	57
AOCEA1	0	27	20	48	90
AOIEA1	3	24	20	49	94
ASSIA2	0	16	11	1013	1025
ASSIB2	0	17	12	2014	2027
ATTRIB	17	42	56	167	348
BALPA1	2	15	10	13	27
BALPA2	2	19	10	15	33
BLEMA2	0	18	9	19	164
BRUAA1	2	30	17	36	86
BRUAA2	2	30	17	38	90
BRUNA1	2	25	16	35	77
BRUNA2	2	26	16	37	79
BSRCA2	10	61	52	117	238
BSRCA3	10	60	53	117	237
C31PA2	94	39	149	433	499
CAPAA1	2	34	20	33	76
CAPAA2	3	37	20	36	83
CAPAB1	2	32	21	35	76
CAPAB2	3	35	22	39	83
CASEA2	2	37	274	1072	1366
CENTA2	0	34	275	300	334
CHSSA1	5	41	42	109	172
CHSSA2	5	42	42	114	179
CPUTIM	0	10	4	9	20
CSBTA1	1	20	13	23	54
CSBTA2	1	24	13	25	60
CSCTA1	1	21	21	39	104
CSCTA2	1	25	21	50	128
CSDTA1	1	21	14	23	56
CSDTA2	1	26	16	29	67
CSETA1	1	21	16	29	74
CSETA2	1	25	16	35	88
CSSTA1	1	21	13	23	56
CSSTA2	1	25	13	26	64
DATABA	136	106	224	754	1780
DRPCA1	4	26	18	37	81
F1IUA1	3	19	15	29	68
F1IUA2	3	19	16	32	71
FACTA1	2	29	13	25	58
FACTA2	2	32	17	35	74
FL2RA1	3	22	14	32	75
FL2RA2	3	23	15	35	79
FLP1A1	2	18	12	21	47
FLP1A2	2	18	13	28	58

Program	n_2*	n_1	n_2	N_2	N_1
FPAAA1	2	23	15	22	49
FPAAA2	3	28	16	24	59
FPAAB1	2	24	18	28	63
FPAAB2	4	30	20	34	85
FPAAC1	2	23	24	40	87
FPAAC2	7	29	29	55	141
FPAAD1	2	23	30	63	138
FPAAD2	12	29	44	93	248
FPANA1	2	18	13	19	42
FPANA2	3	23	14	21	48
FPANB1	2	21	17	27	61
FPANB2	4	25	19	33	75
FPANC1	2	21	23	39	86
FPANC2	7	26	28	54	121
FPAND1	2	24	34	63	138
FPAND2	12	26	43	91	206
FPRAA1	3	30	16	30	76
FPRAA2	3	30	16	32	80
FPRNA1	3	26	16	29	67
FPRNA2	3	26	16	31	69
GVRAA1	3	21	14	23	57
GVRAA2	3	22	14	25	61
GVRNA1	3	19	14	23	51
GVRNA2	3	20	14	25	53
HSDRA2	3	36	36	103	131
IADDA1	1	23	19	37	85
IADDA2	1	24	19	40	88
IDIVA2	1	24	19	40	88
IEXPA2	1	25	19	41	89
IMIXA2	1	27	19	42	92
IMIXB1	1	24	21	51	123
IMIXB2	1	30	21	60	138
IMIXC2	1	27	21	59	137
IMIXD1	1	28	22	64	142
IMIXE1	1	24	21	51	123
IMIXE2	1	31	22	65	141
INQUIR	115	119	248	871	1936
INSTR	111	121	298	1347	2434
INTDA2	0	19	159	462	478
INTDB2	0	14	505	508	1518
INTDB3	0	14	505	508	520
INTQA2	0	26	15	26	58
IOPKG	40	77	126	430	799
ISEQA2	0	28	20	43	87
LAVRA1	2	26	17	34	68
LAVRA2	2	26	17	38	72
LAVRB1	2	26	26	151	320
LAVRB2	2	26	26	191	360
LFIRA1	2	31	21	54	93
LFSRA1	5	26	19	41	84
LOAEA1	2	26	21	41	81

Program	n_2*	n_1	n_2	N_2	N_1
LOECA1	3	24	19	35	66
LOECA2	3	24	19	37	71
LOFCA1	6	28	20	38	85
LOSCA1	3	29	21	45	95
LOUIA1	3	25	18	36	73
LOUIA2	3	25	19	38	74
LRR1A1	2	26	22	39	94
LRR1A2	2	27	22	42	97
LRR3A1	2	26	22	52	113
LRR3A2	2	27	22	57	118
LVRAB1	2	25	23	115	350
LVRAB2	2	25	23	135	390
MINIA2	0	14	6	9	21
MTCQA2	0	23	14	21	49
MTESA2	0	24	18	25	48
MULTA1	0	19	13	20	36
MULTA2	0	20	13	32	48
NL00A1	2	21	17	21	53
NL07A2	2	29	21	39	82
NL65A2	2	29	40	135	340
NPPCA1	4	20	15	31	78
NPPCA2	4	20	15	27	68
NRPCA1	6	30	25	64	129
NRPCA2	6	30	25	64	131
NULLA1	0	18	10	16	35
NULLA2	0	19	10	16	37
OPAEA1	2	28	19	68	139
OPBFA1	1	25	15	39	85
OPCEA1	8	32	20	63	146
OPISA1	2	30	22	90	174
OPNFA1	2	26	15	36	75
OPSCA1	2	35	19	78	126
PGQUA2	0	27	22	42	77
PIALA2	0	23	20	59	94
PKGEA1	26	24	88	242	509
PKGEA2	26	52	88	268	485
PRCOA2	5	59	30	90	173
PUZZA2	26	46	100	562	787
RANDA2	4	31	29	51	103
RCDSA2	0	26	616	4808	7248
RENDA1	1	31	15	32	74
RENDA2	1	32	14	32	71
RPTWRI	8	19	10	12	43
SCHEMA	6	34	233	362	530
SIEVA1	0	22	16	24	53
SIEVA2	3	28	22	54	85
SORTA2	0	30	29	111	159
SQ10A2	0	21	15	23	44
SQPGA2	0	24	17	31	56
SRCRA1	1	35	40	71	125
TAIPA1	2	31	14	31	74

Program	n_2*	n_1	n_2	N_2	N_1
TAIPA2	2	32	16	37	92
TPGTA2	0	32	16	36	83
TPGTC2	20	32	52	125	335
TPITA1	1	30	15	41	93
TPITA2	1	30	16	44	100
TPITB1	1	34	23	77	176
TPITB2	1	37	28	92	211
TPITD1	0	45	52	207	442
TPITD2	0	45	71	266	580
TPOTA2	0	26	40	141	381
TPOTC2	0	125	109	417	114
TPSTA2	0	26	12	29	71
TPTCB2	0	29	19	62	145
TPTCC2	0	34	29	107	245
TPTCD2	0	44	44	220	445
TPUTE2	2	31	16	48	100
UAPAA1	6	40	25	49	113
VFADA1	0	27	17	26	55
VFADA2	0	28	17	32	61
WHETA2	12	43	66	362	444
WHLPA1	3	15	11	15	33
WHLPA2	3	17	11	17	36

APPENDIX E

Where to Begin

1). Getting Access to Computers at AFIT:

a. Obtain and Complete AFIT Form 35 for access to ASC and CSC.

1. Disk Space requirement: 10,000 blocks

2. Enter: man [command] for information on Ada, where command is Ada, a.mklib, a.cleanlib, etc.

b. For access to ISL computer contact professor incharge of system.

1. Obtain and Complete AFIT Form 35, and then give it to the professor in charge of the system.

Current point of contact: Dr. Hartrum

2. For general information on Ada, enter: type sys$doc:ada.doc

3. For help on Ada commands, enter: Help.

2). Getting Access to DG computer:

a. Location - Information Systems and Technology Center, Bldg 676 in Area B.

b. Contact - System Manager

Currently - Capt Deese (Rm. 109, PH. 255-4472)

c. For help on Ada, enter: ADEHELP. (Note: you must be in the ADE environment).

3). Statistical Analysis Tools

a. CSC - SAS statistical package

Note: For information on getting started with SAS enter: type sys$doc:sas.doc.

b. or, SSC - S statistical package

Note: For information on S enter: man S

APPENDIX F

Actual Compile Times

	VAX11/785 UNIX (ASC)	DGMV8000 AOS/VS	VAX11/780 VMS (ISL)	VAX11/785 VMS (CSC)
ADDSA1	3.77	7.88	3.27	1.93
ADDSA2	4.00	9.18	3.54	2.10
AKERA2	4.03	7.49	3.72	2.19
AOCEA1	9.90	12.15	8.14	4.91
AOIEA1	9.60	12.06	8.36	5.12
ASSIA2	12.77	45.48	19.11	10.69
ASSIB2	24.13	93.08	40.03	21.14
ATTRIB	13.23	25.47	14.49	8.30
BALPA1	3.37	5.99	3.33	1.95
BALPA2	3.60	6.36	3.46	2.03
BLEMA2	5.67	8.80	5.01	2.85
BRUAA1	9.03	11.08	8.05	4.84
BRUAA2	9.27	11.29	8.05	4.92
BRUNA1	9.00	10.99	7.89	4.77
BRUNA2	9.17	11.00	7.94	4.74
BSRCA2	13.50	20.35	13.79	8.19
BSRCA3	13.73	20.44	13.78	8.16
C31PA2	13.80	35.25	17.42	10.89
CAPAA1	9.80	11.04	7.80	4.73
CAPAA2	9.73	11.10	7.92	4.78
CAPAB1	10.37	11.00	7.90	4.82
CAPAB2	10.00	11.21	7.96	4.84
CASEA2	43.50	154.12	25.63	14.65
CENTA2	14.30	28.25	7.77	3.72
CHSSA1	7.40	14.06	5.77	3.45
CHSSA2	7.57	14.26	5.94	3.59
CPUTIM	3.78	5.57	5.62	3.39
CSBTA1	8.50	9.79	7.45	4.55
CSBTA2	8.77	9.95	7.54	4.48
CSCTA1	10.00	12.06	8.53	5.14
CSCTA2	10.60	13.23	8.79	5.31
CSDTA1	8.77	9.91	7.49	4.52
CSDTA2	8.60	10.26	7.60	4.53
CSETA1	9.13	10.62	7.85	4.69
CSETA2	9.20	11.06	8.01	4.85
CSSTA1	8.73	10.00	7.50	4.47
CSSTA2	8.67	10.17	7.65	4.54
DATABA	93.73	163.54	119.03	72.89
DRPCA1	9.23	10.87	7.91	4.75
F1IUA1	4.50	7.44	4.27	2.49
F1IUA2	4.67	7.67	4.41	2.66
FACTA1	4.07	6.92	3.62	2.13
FACTA2	5.47	7.69	4.54	2.71
FL2RA1	4.77	7.76	4.28	2.57

	VAX11/785 UNIX (ASC)	DGMV8000 AOS/VS	VAX11/780 VMS (ISL)	VAX11/785 VMS (CSC)
FL2RA2	4.77	8.00	4.41	2.59
FLP1A1	3.67	6.58	3.64	1.69
FLP1A2	4.10	6.83	3.88	2.28
FPAAA1	8.60	9.29	7.32	4.48
FPAAA2	8.87	9.47	7.58	4.56
FPAAB1	9.23	10.07	7.77	4.66
FPAAB2	9.53	10.55	8.15	4.91
FPAAC1	10.33	11.58	8.65	5.23
FPAAC2	11.17	12.54	9.46	5.69
FPAAD1	12.37	14.26	10.24	6.17
FPAAD2	14.33	16.17	11.60	7.10
FPANA1	8.53	9.22	7.30	4.44
FPANA2	8.47	9.27	7.34	4.45
FPANB1	9.30	10.08	7.68	4.68
FPANB2	9.37	10.46	7.96	4.76
FPANC1	10.37	11.47	8.58	5.16
FPANC2	10.83	12.25	9.13	5.47
FPAND1	12.47	14.24	10.17	6.19
FPAND2	13.37	15.45	10.94	6.71
FPRAA1	9.53	10.45	7.86	4.81
FPRAA2	9.50	10.45	7.94	4.77
FPRNA1	9.30	10.29	7.69	4.67
FPRNA2	9.30	10.35	7.75	4.71
GVRAA1	8.93	9.87	7.53	4.47
GVRAA2	8.93	9.94	7.50	4.55
GVRNA1	8.87	9.79	7.40	4.44
GVRNA2	8.57	9.87	7.51	4.53
HSDRA2	8.07	13.95	6.96	4.03
IADDA1	9.47	11.23	8.07	4.82
IADDA2	9.73	11.72	8.02	4.86
IDIVA2	9.40	11.62	8.03	4.88
IEXPA2	9.77	11.50	8.20	4.83
IMIXA2	9.77	12.44	8.08	4.96
IMIXB1	10.93	13.00	8.88	5.28
IMIXB2	11.17	14.98	9.10	5.46
IMIXC2	11.13	14.63	9.06	5.47
IMIXD1	11.10	15.53	9.20	5.60
IMIXE1	10.83	13.01	8.95	5.37
IMIXE2	11.23	15.25	9.16	5.68
INQUIR	84.33	158.19	126.40	80.88
INSTR	59.97	136.58	75.48	40.21
INTDA2	8.47	22.99	13.03	6.95
INTDB2	10.47	35.98	24.60	15.45
INTDB3	9.87	22.91	20.19	14.03
INTQA2	10.13	9.23	6.25	3.86
IOPKG	35.77	81.91	45.93	28.03
ISEQA2	15.40	15.84	10.94	6.81
LAVRA1	11.43	10.86	9.15	5.58

	VAX11/785 UNIX (ASC)	DGMV8000 AOS/VS	VAX11/780 VMS (ISL)	VAX11/785 VMS (CSC)
LAVRA2	11.53	11.11	9.22	5.64
LAVRB1	26.03	22.73	23.48	14.37
LAVRB2	30.40	24.24	24.67	15.15
LFIRA1	10.97	14.23	9.18	5.59
LFSRA1	9.27	11.15	8.21	4.92
LOAEA1	9.20	11.32	8.00	4.86
LOECA1	9.40	10.87	7.96	4.81
LOECA2	9.30	11.21	8.10	4.92
LOFCA1	9.43	11.13	8.27	5.04
LOSCA1	10.27	12.00	9.62	5.84
LOUIA1	9.13	11.08	8.20	4.99
LOUIA2	9.17	11.08	8.20	4.97
LRR1A1	8.97	11.39	8.04	4.78
LRR1A2	9.30	11.46	7.97	4.80
LRR3A1	9.30	12.82	8.36	4.99
LRR3A2	9.37	13.14	8.48	4.98
LVRAB1	16.57	21.01	13.26	8.08
LVRAB2	17.60	22.06	13.71	8.51
MINIA2	3.23	5.08	2.79	1.70
MTCQA2	9.63	8.99	6.06	3.70
MTESA2	11.03	10.90	8.05	5.08
MULTA1	3.97	6.94	3.25	2.00
MULTA2	4.07	8.20	3.52	2.18
NL00A1	9.63	8.60	7.02	4.30
NL07A2	9.63	10.94	8.16	4.93
NL65A2	32.13	46.78	15.15	9.26
NPPCA1	9.63	10.41	7.90	4.82
NPPCA2	9.33	9.95	7.91	4.67
NRPCA1	12.87	12.73	9.35	5.67
NRPCA2	12.87	12.85	9.34	5.71
NULLA1	3.87	6.82	4.47	2.63
NULLA2	3.93	6.76	4.41	2.60
OPAEA1	11.07	15.21	8.98	5.50
OPBFA1	9.60	11.40	8.14	4.95
OPCEA1	11.63	14.14	9.33	5.65
OPISA1	11.83	17.01	9.48	5.68
OPNFA1	8.90	11.21	7.77	4.78
OPSCA1	11.40	14.35	10.20	6.46
PGQUA2	10.03	10.00	6.44	3.99
PIALA2	4.40	9.88	4.26	2.50
PKGEA1	41.70	63.65	82.27	37.55
PKGEA2	43.77	67.57	84.97	40.04
PRCOA2	8.73	12.88	6.73	4.17
PUZZA2	23.10	62.19	20.88	12.52
RANDA2	4.17	8.45	4.01	2.41
RCDSA2	89.87	1535.62	122.16	66.82
RENDA1	5.47	7.67	4.29	2.56
RENDA2	5.30	7.69	4.32	2.62

	VAX11/785 UNIX (ASC)	DGMV8000 AOS/VS	VAX11/780 VMS (ISL)	VAX11/785 VMS (CSC)
RPTWRI	3.70	6.94	2.90	1.77
SCHEMA	74.17	62.97	42.57	26.82
SIEVA1	3.83	6.49	3.54	2.21
SIEVA2	4.63	8.99	4.20	2.54
SORTA2	10.23	15.06	8.75	5.54
SQ10A2	13.50	13.50	9.81	6.26
SQPGA2	13.83	14.15	10.08	6.43
SRCRA1	8.97	14.03	8.97	5.70
TAIPA1	5.43	7.38	4.51	2.83
TAIPA2	5.63	7.64	4.63	2.97
TPGTA2	5.63	7.19	4.63	2.98
TPGTC2	9.47	13.97	8.33	5.27
TPITA1	6.27	7.82	5.05	3.17
TPITA2	6.57	8.00	5.17	3.38
TPITB1	9.93	10.75	7.44	4.78
TPITB2	11.13	11.56	8.09	5.17
TPITD1	23.73	20.98	16.19	10.08
TPITD2	28.80	24.94	18.99	11.92
TPOTA2	12.30	15.99	9.37	5.90
TPOTC2	29.90	42.70	21.76	13.54
TPSTA2	5.50	6.94	4.48	2.80
TPTCB2	8.80	9.39	6.74	4.26
TPTCC2	13.83	13.02	9.87	6.36
TPTCD2	23.10	20.74	16.10	10.24
TPUTE2	6.57	8.40	5.44	3.36
UAPAA1	10.37	12.46	8.53	5.44
VFADA1	59.27	10.07	4.32	2.86
VFADA2	59.43	10.80	4.53	2.96
WHETA2	14.57	42.68	11.73	7.48
WHLPA1	3.73	6.25	3.42	2.15
WHLPA2	3.87	6.58	3.56	2.18

APPENDIX G

Macros Used During the TEST

1. UNIX SYSTEM:

Macro: clr

Code:
```
#
cd  <home directory>/comp_acec
cd .imports
rm *
cd ..
cd .objects
rm *
cd  <home directory>/comp_acec/.nets
rm *
cd  <home directory>/comp_acec/.lines
rm *
cd  <home directory>/comp_acec
rm GVAS_table
rm ada.lib
rm gnrx.lib
cd  <home directory>/begin.lib
cp *.lib <home directory>/comp_acec
cp GVAS* <home directory>/comp_acec
cd .objects
cp * <home directory>/comp_acec/.objects
cd ..
cd .nets
cp * <home directory>/comp_acec/.nets
cd ..
cd .lines
cp * <home directory>/comp_acec/.lines
cd ..
cd .imports
cp * <home directory>/comp_acec/.imports
cd  <home directory>/comp_acec
```

Note: The BEGIN directory contained all libraries necessary for the benchmark test modules to compile, i.e. the standard Ada libraries and IO_PACKAGE, CPU_TIME and INSTRUMENT libraries.

Macro: clrall

Code: Same as above except START.LIB is substituted for BEGIN.LIB

Note: The START directory contained only the standard Ada libraries.

2. AOS/VS SYSTEM:

Macro: clr.cli

```
Code:  dir :accounts:afit:acec
       del +.ob +.str +.tree +.sr +.lst ac+
       dir begin
       move/d ^+
       dir :accounts:afit:acec
```

Note: The BEGIN directory contained all libraries necessary for the benchmark test modules to compile, i.e. the standard Ada libraries and IO_PACKAGE, CPU_TIME and INSTRUMENT libraries.

Macro: clrall.cli

```
Code: dir :accounts:afit:acec
      del +.ob +.str +.tree +.sr +.lst ac+
      dir start
      move/d ^+
      dir :accounts:afit:acec
```

Note: The START directory contained only the standard Ada libraries.

APPENDIX H

SAS Data File for Analysis 1

Name of File: table.sas

```
cards;
ADDSA1  0   19   14   20      36     3.77    7.88     3.27    1.93
ADDSA2  0   19   14   32      48     4.00    9.18     3.54    2.10
AKERA2  3   24   12   28      57     4.03    7.49     3.72    2.19
AOCEA1  0   27   20   48      90     9.90   12.15     8.14    4.91
AOIEA1  3   24   20   49      94     9.60   12.06     8.36    5.12
ASSIA2  0   16   11   1013    1025  12.77   45.48    19.11   10.69
ASSIB2  0   17   12   2014    2027  24.13   93.08    40.03   21.14
ATTRIB 17   42   56   167     348   13.24   25.47    14.49    8.30
BALPA1  2   15   10   13      27     3.37    5.99     3.33    1.95
BALPA2  2   19   10   15      33     5.67    8.80     5.01    2.85
BLEMA2  0   18   9    19      164    9.03   11.08     8.05    4.84
BRUAA1  2   30   17   36      86     9.27   11.27     8.05    4.92
BRUAA2  2   30   17   38      90     9.00   10.99     7.89    4.77
BRUNA1  2   25   16   35      77     9.17   11.00     7.94    4.74
BRUNA2  2   26   16   37      79    13.50   20.35    13.79    8.19
BSRCA2 10   61   52   117     238   13.73   20.44    13.78   10.89
BSRCA3 10   60   53   117     237    9.80   11.04     7.80    4.73
                              .
                              .
                             etc
```

where

Column 1 - Program
Column 2 - n_2*
Column 3 - n_1
Column 4 - n_2
Column 5 - N_2
Column 6 - N_1
Column 7 - Unix Compile Times
Column 8 - AOS/VS Compile Times
Column 9 - VMS-ISL Compile Times
Column 10 - VMS-CSC Compile Times

NOTE: The data file must have a '.sas' suffix and at least one blank space between columns.

APPENDIX I

SAS Command Files for Analysis 1

Name of File: Test1.sas

Objective: Determine the adjusted coefficient of determination for each model and estimate the unknown parameters for Model 1.

Code:

```
DATA;
INPUT ID $ IO N1 N2 CN2 CN1 UNIX AOSVS ISL CSC;
Ntotal = CN1 + CN2;
Nhat = N1 * LOG2(N1) + N2 * LOG2(N2);
LOGNtot = LOG(Ntotal);
LOGNhat = LOG(Nhat);
VOL = Ntotal * LOG2(2 + N2);
Vstar = (2 + IO) * LOG2(2 + IO);
LOGVOL = LOG(VOL);
LOGVstar = LOG(Vstar);
VOLest = Nhat * LOG2(N1 + N2);
Lhat = 2/N1*N2/CN2;
LOGVOLes = LOG(VOLest);
LOGLhat = LOG(Lhat);
logtime1 = LOG(UNIX);
logtime2 = LOG(AOSVS);
logtime3 = LOG(ISL);
logtime4 = LOG(CSC);
Effort = VOL**2 / (Vstar);
%INCLUDE table;
PROC REG;
 Model UNIX = Ntotal;
 Model UNIX = Nhat;
 Model AOSVS = Ntotal;
        .
        .
 Model CSC = Nhat;
 Model logtime1 = LOGNtot;
 Model logtime1 = LOGNhat;
 Model logtime2 = LOGNtot;
        .
        .
 Model logtime4 = LOGNhat;
 Model logtime1 = LOGVOL LOGVstar;
 Model logtime1 = LOGVOLes LOGLhat;
 Model logtime2 = LOGVOL LOGVstar;
        .
        .
 Model logtime4 = LOGVOLES LOGLhat;
 Model UNIX = Effort;
 Model AOSVS = Effort;
 Model ISL = Effort;
 Model CSC = Effort;
```

Name of File: Test2.sas

Objective: Obtain predicted times and correlate predicted and actual compile times for Model 1.

Code:

```
DATA;
INPUT ID $ IO N1 N2 CN2 CN1 UNIX AOSVS ISL CSC;
Ntotal = CN1 + CN2;
VOL = Ntotal * LOG2(N1 + N2);
Vstar = (2 + IO) * LOG2(2 + IO);
PUNIX = .5281 * (VOL**.4124);
PAOSVS = .2216 * (VOL**.5830 * Vstar**.0431);
PISL = .2641 * (VOL**.4730 * Vstar**.1047);
PCSC = .1681 * (VOL**.4670 * Vstar**.0991);
RUNIX = Unix - PUNIX;
RAOSVS = AOSVS - PAOSVS;
RISL = ISL - PISL;
RCSC = CSC - PCSC;
%INCLUDE table;
PROC PRINT;
PROC CORR;
 VAR UNIX PUNIX AOSVS PAOSVS ISL PISL CSC PCSC;
```

NOTE: 'P' prefix - Predicted Compile Time
'R' prefix - Residual

Execution: runsas [filename] <New Line>
<New line>

Note: The '.sas' suffix not required.

APPENDIX J

SAS Data File for Analysis 2

Name of File: table1.sas

```
cards;
ADDSA1 0    19     14     20     36     3.77   0  0
ADDSA2 0    19     14     32     48     4.00   0  0
AKERA2 3    24     12     28     57     4.03   0  0
AOCEA1 0    27     20     48     90     9.90   0  0
AOIEA1 3    24     20     49     94     9.60   0  0
                          .
                          .
                         etc
ADDSA1 0    19     14     20     36     7.88   0  1
ADDSA2 0    19     14     32     48     9.18   0  1
AKERA2 3    24     12     28     57     7.49   0  1
AOCEA1 0    27     20     48     90    12.15   0  1
AOIEA1 3    24     20     49     94    12.06   0  1
                          .
                          .
                         etc
ADDSA1 0    19     14     20     36     3.27   1  0
ADDSA2 0    19     14     32     48     3.54   1  0
AKERA2 3    24     12     28     57     3.72   1  0
AOCEA1 0    27     20     48     90     8.14   1  0
AOIEA1 3    24     20     49     94     8.36   1  0
                          .
                          .
                         etc
ADDSA1 0    19     14     20     36     1.93   1  1
ADDSA2 0    19     14     32     48     2.10   1  1
AKERA2 3    24     12     28     57     2.19   1  1
AOCEA1 0    27     20     48     90     4.91   1  1
AOIEA1 3    24     20     49     94     5.12   1  1
                          .
                          .
                         etc
```

where

Column 1 - Program
Column 2 - n_2*
Column 3 - n_1
Column 4 - n_2
Column 5 - N_2
Column 6 - N_1
Column 7 - Compile Times for all Computers
Column 8 - Dummy Variable 'e'
Column 9 - Dummy Varaible 'f'

NOTE: The data file must have a '.sas' suffix and at least one blank space between columns.

APPENDIX K

SAS Command Files for Analysis 2

Name of File: Test3.sas

Objective: Determine the translation rate for each compiler.

Code:
```
DATA;
INPUT ID $ IO N1 N2 CN2 CN1 TIMES A B;
Ntotal = CN1 + CN2;
VOL = Ntotal * LOG2(2 + N2);
Vstar = (2 + IO) * LOG2(2 + IO);
LOGVOL = LOG(VOL);
LOGVstar = LOG(Vstar);
Logtime = LOG(TIMES);
%INCLUDE table1;
PROC REG;
 Model Logtime = LOGVOL LOGVstar;
```

Name of File: Test3.sas

Objective: Obtain compile times from the linear equation for each compiler.

Code:
```
DATA;
INPUT ID $ IO N1 N2 CN2 CN1 UNIX AOSVS ISL CSC;
Ntotal = CN1 + CN2;
VOL = Ntotal * LOG2(N1 + N2);
Vstar = (2 + IO) * LOG2(2 + IO);
LOGUNIX = -.9918 + (.4839*LOGVOL)+(.0745*LOGVstar);
LOGAOSVS = -1.0881 + (.4839*LOGVOL)+(.0745*LOGVstar);
LOGISL = -1.5419 + (.4839*LOGVOL)+(.0745*LOGVstar);
LOGCSC = -1.6482 + (.4839*LOGVOL)+(.0745*LOGVstar);
%INCLUDE table;
PROC SORT;
 by UNIX;
PROC Print;
 VAR LOGUNIX LOGAOSVS LOGISL LOGCSC;
```

Execution: runsas [filename] <New Line>
<New line>

Note: The '.sas' suffix not required.

APPENDIX L

Sample SAS Output

SAS Output:

SAS

DEP VARIABLE: LOGTIME2

ANALYSIS OF VARIANCE

SOURCE	DF	SUM OF SQUARES	MEAN SQUARE	F VALUE	PROB>F
MODEL	2	77.92752	38.96376	444.600	0.0001
ERROR	168	14.72316	O.08763785		
C TOTAL	170	92.65068			

ROOT MSE	0.2960369	R-SQUARE	0.8411
DEP MEAN	2.606862	ADJ R-SQ	0.8392

PARAMETER ESTIMATES

VARIABLE	DF	PARAMETER ESTIMATE	STANDARD ERROR	T FOR HO: PARAMTER=0	PROB>\T\
INTERCEP	1	-1.51681	0.1413134	-10.734	0.0001
LOGVOL	1	0.5838089	0.02141666	27.260	0.0001
LOGVSTAR	1	0.04561004	0.01981437	2.302	0.0226

The ANALYSIS OF VARIANCE part of the printout displays results of tests to determine if the model is significant. In this example, the model being tested is:

$$TIME = K * V^a * (V^z)^b,$$

and the null hypothesis is:

V and V^z are NOT significant in computing TIME.

There are four values (21:690-691) in this portion of the printout that are of particular interest to this analysis.

1) The number under the entry F VALUE is the F Value for testing the hypothesis that all parameters are zero except for the intercept. If this number is near 1, the null hypothesis can be accepted. If this number is large, the null hypothesis can be rejected and it can be concluded that the model as a whole is significant.

2) The number under the entry PROB>F is the probability of getting a greater F statistic than that observed if the hypothesis is true. This is the significance probability. In this example, there is a 99.99 probability that the null hypothesis is FALSE, and therefore the null hypothesis can be rejected and it can be concluded that V and V^2 are significant in computing TIME.

3) The number to the right of the R-SQUARE entry is a measure between 0 and 1 that indicates the portion of the (corrected) total variation that is attributed to the fit rather than left to residual error. This value is also called the coefficient of determination. It is the square of the correlation between the dependent variable and the predicted values. This value is not used in this research because it cannot be used to compare models that have different degrees of freedom.

4. The number to the right of the ADJ R-SQ entry is an adjusted version of R-SQUARE that has been adjusted for degrees of freedom. Because this value is adjusted for

degrees of freedom, it can be used to compare different models. In this example, the ADJ R-SQ tells us that we can account for over 83.92 percent of the compile time just by knowing V and V*. We can also conclude that this model reduces the error in estimating compilation time by 83.92 percent over the average compile time.

The PARAMETER ESTIMATES part of the printout displays the estimated values of unknown variables and also displays the results of tests to determine if those variables are significant in the overall model. In this example, the estimated linear equation is:

$$LOG(TIME) = -1.51681 + 0.584LOG(V) + 0.046LOG(V^*);$$

Note: LOG = natural logarithm.

which is equivalent to:

$$Time = (e^{-1.51681}) * V^{(0.584)} * (V^*)^{0.046}.$$

There are also four values in this part of the printout that are of interest:

1) The first value under the PARAMETER ESTIMATES entry is the estimated value for the first variable, INTERCEP.

2) The first value under the PROB>\T\ is also associated with the first variable, INTERCEP. This value is the probability that a T statistic would obtain a greater absolute value than that observed given that the true parameter is zero. This is the two-tailed significance

probability. If this number is small the value of the first parameter is not likely to be 0; therefore, the parameter contributes significantly to the overall model.

3) The second and third values under the PARAMETER ESTIMATES entry are the estimated values for the exponents for V and V*.

4) The second and third values under the PROB>\T\ entry are the T statistics test values for V and V*. As in 2 above, if this number is small, the value of the associated parameter is not likely to be 0; therefore, the parameter contributes significantly to the overall model.

APPENDIX M

Actual (A) vs Predicted (P) Times

	UNIX		AOS/VS		VMS-ISL		VMS-CSC	
Program	A	P	A	P	A	P	A	P
ADDSA1	3.77	5.41	7.88	6.13	3.27	4.09	1.93	2.51
ADDSA2	4.00	6.27	9.18	7.54	3.54	4.85	2.10	2.97
AKERA2	4.03	6.49	7.49	8.56	3.72	6.07	2.19	3.68
AOCEA1	9.90	8.17	12.15	10.97	8.14	6.57	4.91	4.00
AOIEA1	9.60	8.23	12.06	11.96	8.36	7.97	5.12	4.81
ASSIA2	12.77	23.26	45.48	48.15	19.11	21.82	10.69	13.09
ASSIB2	24.13	31.13	93.08	72.67	40.03	30.47	21.14	18.21
ATTRIB	13.23	15.11	25.47	30.70	14.49	19.60	8.30	11.59
BALPA1	3.37	4.55	5.99	5.10	3.33	3.89	1.95	2.37
BALPA2	3.60	5.00	6.36	5.82	3.46	4.33	2.03	2.64
BLEMA2	5.67	8.61	8.80	11.81	5.01	6.98	2.85	4.25
BRUAA1	9.03	7.77	11.08	10.84	8.05	7.17	4.84	4.34
BRUAA2	9.27	7.92	11.29	11.15	8.05	7.33	4.92	4.43
BRUNA1	9.00	7.39	10.99	10.10	7.89	6.77	4.77	4.10
BRUNA2	9.17	7.51	11.00	10.34	7.94	6.90	4.74	4.18
BSRCA2	13.50	13.13	20.35	24.48	13.79	15.61	8.19	9.29
BSRCA3	13.73	13.12	20.44	24.44	13.78	15.59	8.16	9.27
C31PA2	13.80	20.39	35.25	51.22	17.42	34.27	10.89	19.95
CAPAA1	9.80	7.52	11.04	10.36	7.80	6.91	4.73	4.18
CAPAA2	9.73	7.84	11.10	11.17	7.92	7.54	4.78	4.55
CAPAB1	10.37	7.56	11.00	10.44	7.90	6.95	4.82	4.21
CAPAB2	10.00	7.92	11.21	11.33	7.96	7.63	4.84	4.60
CASEA2	43.50	31.49	154.12	78.41	25.63	35.70	14.65	21.16
CENTA2	14.30	18.06	28.25	33.66	7.77	16.32	3.72	9.83
CHSSA1	7.40	11.59	14.06	19.86	5.77	12.47	3.45	7.46
CHSSA2	7.57	11.81	14.26	20.40	5.94	12.74	3.59	7.62
CPUTIM	3.78	3.68	5.57	3.55	5.62	2.63	3.39	1.62
CSBTA1	8.50	6.17	9.79	7.66	7.45	5.22	4.55	3.18
CSBTA2	8.77	6.52	9.95	8.27	7.54	5.55	4.48	3.38
CSCTA1	10.00	8.19	12.06	11.43	8.53	7.22	5.14	4.37
CSCTA2	10.60	9.05	13.23	13.17	8.79	8.09	5.31	4.90
CSDTA1	8.77	6.28	9.91	7.85	7.49	5.32	4.52	3.24
CSDTA2	8.60	6.95	10.26	9.06	7.60	5.98	4.53	3.63
CSETA1	9.13	7.05	10.62	9.25	7.85	6.08	4.69	6.69
CSETA2	9.20	7.68	11.06	10.43	8.01	6.70	4.85	4.07
CSSTA1	8.73	6.26	10.00	7.82	7.50	5.30	4.47	3.23
CSSTA2	8.67	6.69	10.17	8.59	7.65	5.72	4.54	3.48
DATABA	93.73	32.13	163.54	99.26	119.03	60.44	72.89	34.87
DRPCA1	9.23	7.61	10.87	10.82	7.91	7.50	4.75	4.52
F1IUA1	4.50	6.81	7.44	9.15	4.27	6.41	2.49	3.88
F1IUA2	4.67	7.01	7.67	9.53	4.41	6.63	2.66	4.01
FACTA1	4.07	6.55	6.92	8.51	3.62	5.89	2.13	3.57
FACTA2	5.47	7.45	7.69	10.21	4.54	6.83	2.71	4.14
FL2RA1	4.77	7.14	7.66	9.78	4.28	6.77	2.57	4.09
FL2RA2	4.77	7.38	8.00	10.24	4.41	7.03	2.59	4.25
FLP1A1	3.67	5.80	6.58	7.17	3.64	5.13	1.69	3.12

	UNIX		AOS/VS		VMS-ISL		VMS-CSC	
Program	A	P	A	P	A	P	A	P
FLP1A2	4.10	6.41	6.83	8.27	3.88	5.76	2.28	3.49
FPAAA1	8.60	6.07	9.29	7.65	7.32	5.40	4.48	3.28
FPAAA2	8.87	6.58	9.47	8.71	7.58	6.16	4.56	3.73
FPAAB1	9.23	6.80	10.07	8.98	7.77	6.15	4.66	3.73
FPAAB2	9.53	7.74	10.55	11.09	8.15	7.65	4.91	4.61
FPAAC1	10.33	7.90	11.58	11.10	8.65	7.31	5.23	4.42
FPAAC2	11.17	9.65	12.54	15.57	9.46	10.51	5.69	6.29
FPAAD1	12.37	9.66	14.26	14.76	10.24	9.21	6.17	5.55
FPAAD2	14.33	12.40	16.17	22.81	11.60	14.96	7.10	8.90
FPANA1	8.53	5.57	9.22	6.77	7.30	4.89	4.44	2.97
FPANA2	8.47	5.98	9.27	7.61	7.34	5.52	4.45	3.35
FPANB1	9.30	6.63	10.08	8.67	7.68	5.98	4.68	3.63
FPANB2	9.37	7.33	10.46	10.28	7.96	7.19	4.76	4.34
FPANC1	10.37	7.79	11.47	10.88	8.58	7.19	5.16	4.35
FPANC2	10.83	9.15	12.25	14.42	9.13	9.88	5.47	5.92
FPAND1	12.47	9.75	14.24	14.96	10.17	9.31	6.19	5.61
FPAND2	13.37	11.66	15.45	20.88	10.94	13.93	6.71	8.29
FPRAA1	9.53	7.31	10.45	10.11	7.86	6.96	4.81	4.20
FPRAA2	9.50	7.48	10.45	10.44	7.94	7.14	4.77	4.31
FPRNA1	9.30	6.95	10.29	9.41	7.69	6.56	4.67	3.97
FPRNA2	9.30	7.07	10.35	9.64	7.75	6.69	4.67	4.04
GVRAA1	8.93	6.32	9.87	8.22	7.53	5.88	4.47	3.56
GVRAA2	8.93	6.53	9.94	8.62	7.50	6.11	4.55	3.70
GVRNA1	8.87	6.07	9.79	7.78	7.40	5.62	4.44	3.41
GVRNA2	8.57	6.23	9.87	8.06	7.51	5.79	4.53	3.50
HSDRA2	8.07	10.64	13.95	17.18	6.96	10.69	4.03	6.43
IADDA1	9.47	7.67	11.23	10.42	8.07	6.69	4.82	4.06
IADDA2	9.73	7.85	11.72	10.75	8.02	6.87	4.86	4.17
IDIVA2	9.40	7.85	11.62	10.75	8.03	6.87	4.88	4.17
IEXPA2	9.77	7.92	11.50	10.89	8.20	6.94	4.83	4.21
IMIXA2	9.77	7.05	12.44	11.16	8.08	7.08	4.96	4.29
IMIXB1	10.93	8.95	13.00	12.95	8.88	7.99	5.28	4.84
IMIXB2	11.17	9.57	14.98	14.23	9.10	8.62	5.46	5.21
IMIXC2	11.13	9.46	14.63	14.02	9.06	8.52	5.47	5.15
IMIXD1	11.10	9.70	15.53	14.52	9.20	8.76	5.60	5.30
IMIXE1	10.83	8.95	13.01	12.95	8.95	7.99	5.37	4.84
IMIXE2	11.23	9.76	15.25	14.64	9.16	8.82	5.68	5.34
INQUIR	84.33	33.77	158.19	105.57	126.40	62.67	80.88	36.17
INSTR	59.97	38.53	136.58	127.00	75.48	72.59	40.21	41.82
INTDA2	8.47	20.38	22.99	39.92	13.03	18.74	6.95	11.27
INTDB2	10.47	30.22	35.98	69.69	24.60	29.45	15.45	17.61
INTDB3	9.87	22.84	22.91	46.93	20.19	21.36	14.03	12.83
INTQA2	10.13	6.56	9.23	8.04	6.25	5.11	3.86	3.12
IOPKG	35.77	22.99	81.91	58.07	45.93	35.32	28.03	20.65
ISEQA2	15.40	7.99	15.84	10.63	10.94	6.41	6.81	3.90
LAVRA1	11.43	7.14	10.86	9.63	9.15	6.51	5.58	3.95
LAVRA2	11.53	7.37	11.11	10.07	9.22	6.75	5.64	4.09
LAVRB1	26.03	13.70	22.73	24.19	23.48	13.75	14.37	8.25
LAVRB2	30.40	14.62	24.24	26.50	24.67	14.81	15.15	8.88
LFIRA1	10.97	8.48	14.23	12.27	9.18	7.93	5.59	4.79
LFSRA1	9.27	7.81	11.15	11.35	8.21	7.92	4.92	4.77

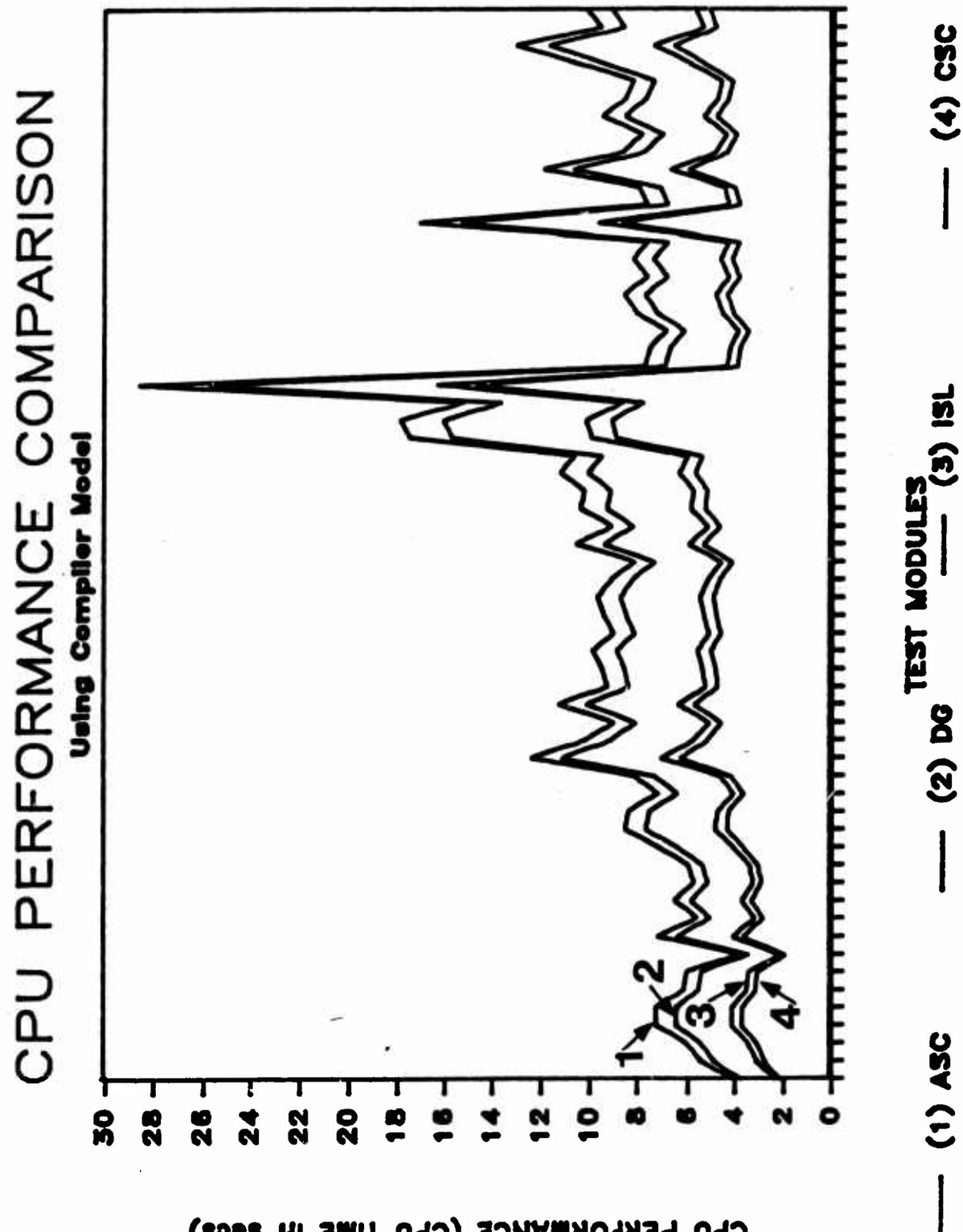
Program	UNIX A	UNIX P	AOS/VS A	AOS/VS P	VMS-ISL A	VMS-ISL P	VMS-CSC A	VMS-CSC P
LOAEA1	9.20	7.77	11.32	10.84	8.00	7.17	4.86	4.34
LOECA1	9.40	7.12	10.87	9.73	7.96	6.74	4.81	4.07
LOECA2	9.30	7.31	11.21	10.12	8.10	6.96	4.92	4.20
LOFCA1	9.43	7.81	11.13	11.46	8.27	8.09	5.04	4.87
LOSCA1	10.27	8.27	12.00	12.05	9.62	8.01	5.84	4.83
LOUIA1	9.13	7.34	11.08	10.17	8.20	6.99	4.99	4.22
LOUIA2	9.17	7.44	11.08	10.37	8.20	7.10	4.97	4.29
LRR1A1	8.97	8.07	11.39	11.43	8.04	7.49	4.78	4.53
LRR1A2	9.30	8.23	11.46	11.77	7.97	7.66	4.80	4.63
LRR3A1	9.30	8.82	12.82	12.97	8.36	8.29	4.99	5.01
LRR3A2	9.37	9.05	13.14	13.46	8.48	8.55	4.98	5.16
LVRAB1	16.57	13.52	21.01	23.72	13.26	13.53	8.08	8.12
LVRAB2	17.60	14.21	22.06	25.46	13.71	14.33	8.51	8.59
MINIA2	3.23	3.93	5.08	3.89	2.79	2.84	1.70	1.75
MTCQA2	9.63	6.02	8.99	7.11	6.06	4.62	3.70	2.83
MTESA2	11.03	6.21	10.90	7.44	8.05	4.79	5.08	2.93
MULTA1	3.97	5.39	6.94	6.10	3.25	4.08	2.00	2.50
MUTLA2	4.07	6.27	8.20	7.55	3.52	4.85	2.18	2.97
NL00A1	9.63	6.17	8.60	7.83	7.02	5.51	4.30	3.34
NL07A2	9.63	7.79	10.94	10.89	8.16	7.19	4.93	4.35
NL65A2	32.13	14.15	47.78	25.31	15.15	14.26	9.26	8.55
NPPCA1	9.63	7.17	10.41	9.97	7.90	7.01	4.82	4.23
NPPCA2	9.33	6.78	9.95	9.20	7.91	6.57	4.67	3.97
NRPCA1	12.87	9.54	12.73	15.20	9.35	10.18	5.67	6.10
NRPCA2	12.97	9.58	12.85	15.29	9.34	10.23	5.71	6.13
NULLA1	3.81	5.11	6.82	5.64	4.47	3.83	2.63	2.35
NULLA2	3.93	5.21	6.76	5.81	4.41	3.92	2.60	2.41
OPAEA1	11.07	9.66	15.21	14.75	8.98	9.20	5.50	5.55
OPBFA1	9.60	7.68	11.40	10.44	8.14	6.70	4.95	4.07
OPCEA1	11.63	9.80	14.14	16.01	9.33	10.87	5.65	6.50
OPISA1	11.83	10.79	17.01	17.26	9.48	10.45	5.68	6.29
OPNFA1	8.90	7.36	11.21	10.04	7.77	6.74	4.78	4.08
OPSCA1	11.40	9.74	14.35	14.93	10.20	9.30	6.46	5.61
PGQUA2	10.03	7.72	10.00	10.13	6.44	6.16	3.99	3.76
PIALA2	4.40	8.44	9.88	11.49	4.26	6.82	2.50	4.16
PKGEA1	41.70	17.87	63.65	39.76	82.27	25.05	37.55	14.74
PKGEA2	43.77	18.23	67.57	40.91	84.97	25.64	40.04	15.08
PRCOA2	8.73	11.36	12.88	19.28	6.73	12.18	4.17	7.29
PUZZA2	23.10	23.27	62.19	57.76	28.80	33.92	12.52	19.88
RANDA2	4.17	8.77	8.45	13.24	4.01	8.83	2.41	5.31
RCDSA2	89.87	63.93	****	201.02	122.16	69.56	66.82	41.13
RENDA1	5.47	7.31	7.67	9.73	4.29	6.33	2.56	3.85
RENDA2	5.30	7.23	7.69	9.57	4.32	6.25	2.62	3.80
RPTWRI	3.70	5.29	6.94	6.70	2.90	5.36	1.77	3.23
SCHEMA	74.17	20.57	62.97	45.03	42.57	24.58	26.82	14.57
SIEVA1	3.83	6.27	6.49	7.55	3.54	4.85	2.21	2.97
SIEVA2	4.63	8.25	8.99	12.00	4.20	7.99	2.54	4.82

NOTE: **** = 1535.62

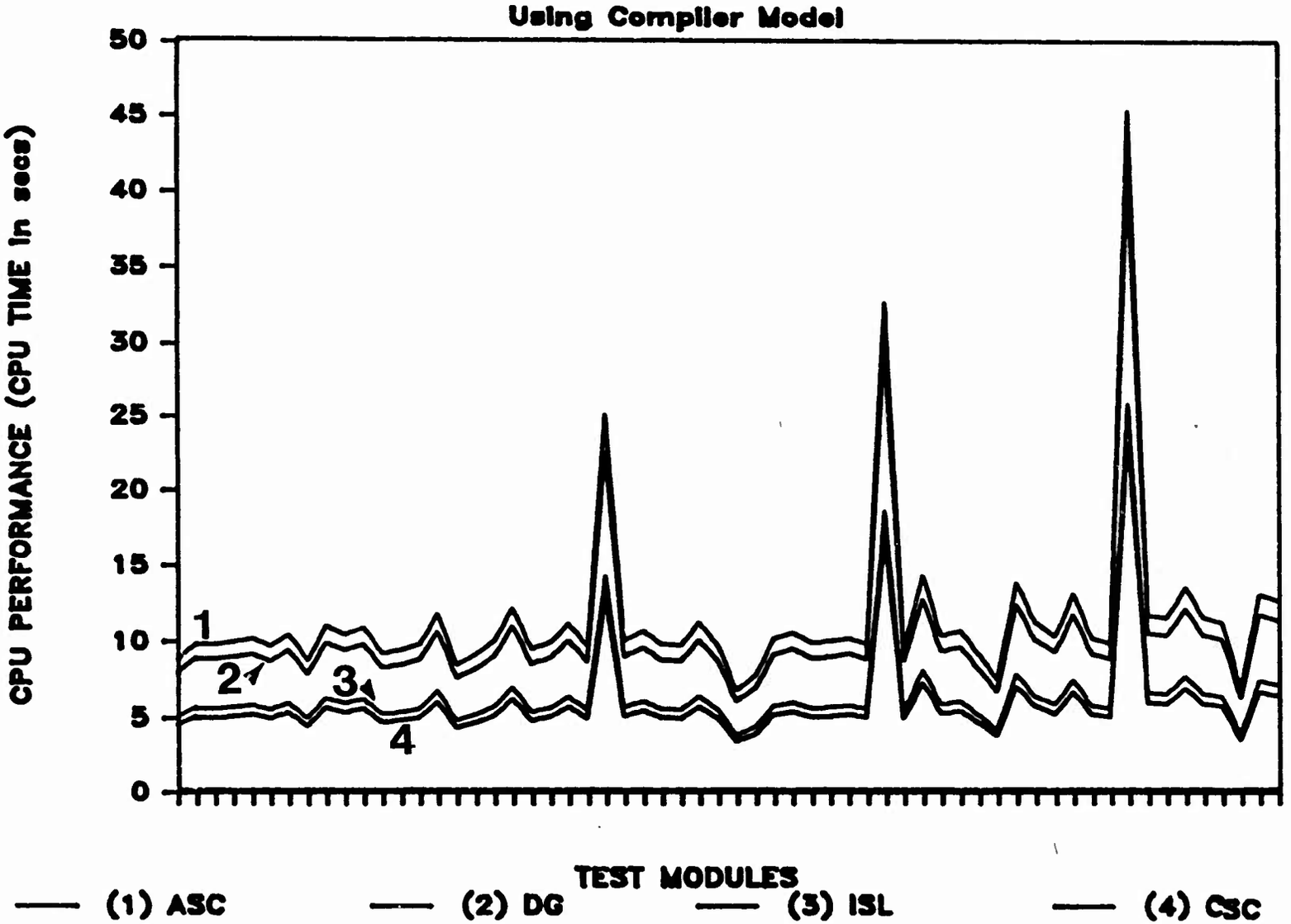
	UNIX		AOS/VS		VMS-ISL		VMS-CSC	
Program	A	P	A	P	A	P	A	P
SORTA2	10.23	11.04	15.06	16.78	8.75	9.28	5.54	5.63
SQ10A2	13.50	5.89	13.50	6.90	9.81	4.51	6.26	2.76
SQPGA2	13.83	6.66	14.15	8.21	10.09	5.19	6.43	3.17
SRCRA1	8.97	9.90	14.03	14.94	8.97	8.97	5.70	5.42
TAIPA1	5.43	7.27	7.38	9.87	4.51	6.64	2.83	4.02
TAIPA2	5.63	7.97	7.64	11.23	4.63	7.38	2.97	4.46
TPGTA2	5.63	7.70	7.19	10.10	4.63	6.14	2.98	3.75
TPGTC2	9.47	14.23	13.97	28.41	8.33	18.66	5.27	11.03
TPITA1	6.27	8.04	7.82	11.12	5.05	7.06	3.17	4.28
TPITA2	6.57	8.30	8.00	11.64	5.17	7.32	3.38	4.44
TPITB1	9.93	10.71	10.75	16.68	7.44	9.81	4.78	5.92
TPITB2	11.13	11.69	11.56	18.88	8.09	10.84	5.17	6.54
TPITD1	23.73	16.61	20.98	29.91	16.19	14.83	10.08	8.94
TPITD2	28.80	18.83	24.94	35.70	18.99	17.12	11.92	10.30
TPOTA2	12.30	14.65	15.99	25.03	9.37	12.83	5.90	7.75
TPOTC2	29.90	25.63	42.70	55.21	21.76	24.38	13.54	14.61
TPSTA2	5.50	6.99	6.94	8.80	4.48	5.49	2.80	3.35
TPTCB2	8.80	9.68	9.39	13.94	6.74	7.98	4.26	4.85
TPTCC2	13.83	12.39	13.02	19.76	9.87	10.59	6.36	6.42
TPTCD2	23.10	16.63	20.74	29.96	16.10	14.85	10.24	8.95
TPUTE2	6.57	8.41	8.40	12.13	5.44	7.85	3.36	4.75
UAPAA1	10.37	9.02	12.46	14.05	8.53	9.55	5.44	5.73
VFADA1	59.27	6.51	10.07	7.96	4.32	5.07	2.86	3.10
VFADA2	59.43	6.91	10.80	8.66	4.53	5.42	2.96	3.31
WHETA2	14.57	18.36	42.68	39.68	11.73	23.45	7.48	13.86
WHLPA1	3.73	4.93	6.25	5.80	3.42	4.43	2.15	2.69
WHLPA2	3.87	5.19	3.58	6.23	3.56	4.69	2.18	2.85

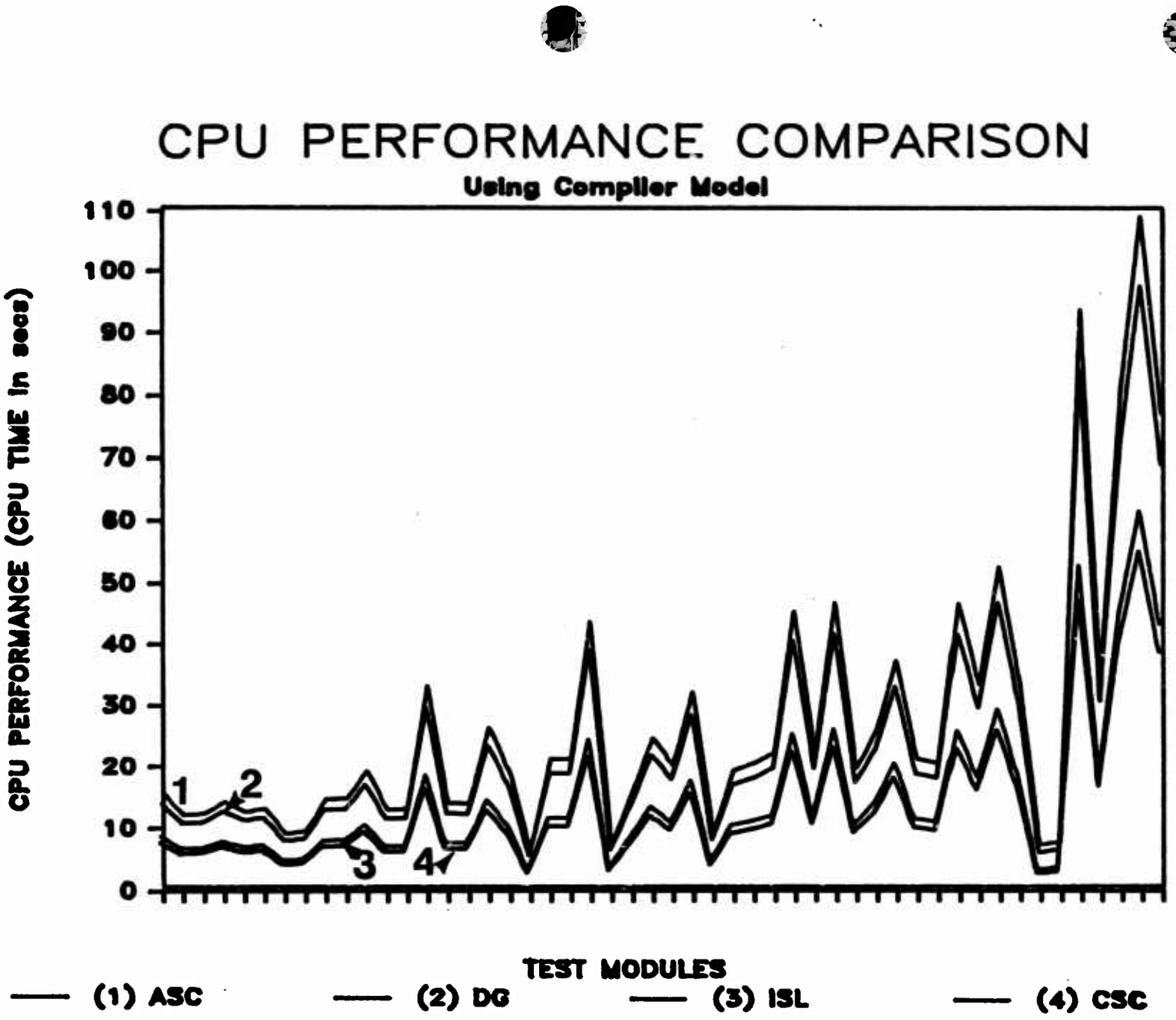
APPENDIX N

Plot of the Actual vs Predicted Compile Times



CPU PERFORMANCE COMPARISON




CPU PERFORMANCE COMPARISON
Using Compiler Model
CPU PERFORMANCE (CPU TIME in secs)
110
100
90
80
70
60
50
40
30
20
10
0
1
2
3
4
TEST MODULES
(1) ASC
(2) DG
(3) ISL
(4) CSC

Bibliography

1. Barnes, J. G. P. Programming in Ada. London: Addison-Wesley Publishing Company, 1984.

2. Beser, N. "Foundations and Experiments in Software Science," ACM SIGMETRICS Performance Evaluation Review, 1980, pp 773-779.

3. Boehm, B. W. "Software Life Cycle Factors", Handbook of Software Engineering, edited by C. R. Vick and C. V. Ramamoorthy. New York: Van Nostrand Reinhold Company, 1984.

4. Booch, Grady. Software Engineering with Ada. Menlo Park, California: The Benjamin/Cummings Publishing Company, 1983.

5. Conte, S. D., V. Y. Shen and K. Dickey. "On the Effect of Different Counting Rules for Control Flow Operators on Software Science Metrics in FORTRAN," Performance Evaluation Review, 11 (2): (Summer 1982).

6. Coulter, N. S. "Software Science and Cognitive Psychology," IEEE Trans. Software Engineering, SE-9, 2(1983), pp 166-171.

7. Department of Defense. Military Standard: Ada Programming Language - ANSI/MIL-STD-1815A. Washington, D.C., January 1983.

8. Elshoff, J. L. "An Investigation into the Effects of the Counting Method Used on Software Science Measurements," ACM SIGPLAN Notices, 13 (2): 30-45 (February 1978).

9. Fairley, Richard E. Software Engineering Concepts. New York: McGraw-Hill Book Company, 1985.

10. Fitzsimmons, Ann and Tom Love. "A Review and Evaluation of Software Science," ACM Computer Surveys, 10 (1): 3-17 (March 1978).

11. Halstead, Maurice H. Elements of Software Science. New York: Elsevier North_Holland Inc., 1977.

12. Hook, Audrey A. and others. "User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC) Version 1", IDA Paper P-1879, (October 1985).

13. Kavi, Krisha M. and U. B. Jackson. "Effect of Declarations on Software Metrics," Performance Evaluation Review, 11 (2): (Summer 1982).

14. Kolence, Kenneth W. An Introduction to Software Physics. New York: McGraw-Hill Book Company, 1985.

15. Leathrum, J. F. Foundation of Software Design. Reston Virginia: Reston Publishing Company, Inc, 1983.

16. Lee, R. C. T. "Compilers", Handbook of Software Engineering, edited by C. R. Vick and C. V. Ramamoorthy. New York : Van Nostrand Reinhold Company, 1984.

17. Maness, Capt Robert S. Validation of a Structural Model of Computer Compile Time. MS Thesis, GCS/ENG/86D. School of Engineering, Air Force Institute of Technology, (AU), Wright-Patterson AFB OH, December 1986.

18. McClave, James T. and P. George Benson. Statistics for Business and Economics: Second Edition. San Francisco and Santa Clara, California: Dellen Publishing Company, 1982.

19. Misek-Falkoff, Linda D. "A Unification of Halstead's Software Science Counting Rules for Programs and English Text, and a Claim Space Approach to Extensions," Performance Evaluation Review, 11 (2): 80-114 (Summer 1982).

20. Relph, Richard, Steve Hahn, and Fred Viles. "Benchmarking C Compilers," Dr. Dobb's Journal of Software Tools: 11 (8): 30-50 (August 1986).

21. Salt, Norman F. "Defining Software Counting Strategies," ACM SIGPLAN Notices, 17 (3): 58-67 (March 1982).

22. SAS Institute Inc. SAS User's Guide: Statistics Version 5 Edition. Cary, N.C.: SAS Institute Inc, 1985 956 pp.

23. Shen, Vincent Y., Samuel D. Conte, and H. E. Dunsmore. "Software Science Revisted: A Critical Analysis of the Theory and its Empirical Support," IEEE Trans. Software Engineering, SE-9, 2 (1983), pp 155-165.

24. Trembly, Jean-Paul and Paul G. Sorenson. The Theory and Practice of Compiler Writing. New York: McGraw-Hill Book Company, 1985.

25. Witt, Donald J. Using Ada in the Real-Time Avionics Environment: Issues and Conclusions. MS Thesis GCS/MA/85D-6. School of Engineering, Air Force Institute of Technology, (AU), Wright-Paterson AFB OH, December 1985.

26. Wolverton, R.W. "Software Costing," Handbook of Software Engineering, edited by C. R. Vick and C. V. Ramamoorthy. New York: Van Nostrand Reinhold Company, 1984.

27. Young, S. J. An Introduction to Ada. Chichester, England: Ellis Horwood Limited, 1983.

28. Zweben, S. and K. Fung. "Exploring Software Science Relations in COBOL and APL," Proceedings, COMPSAC, 1979, 702-707.

VITA

Dennis M. Miller [redacted]. He graduated from high school in Wiesbaden, West Germany in 1976 and attended North Dakota State University, Fargo, North Dakota, from which he received the degree of Bachelor of Science in Electrical and Electronic Engineering in May 1981. Upon graduation, he received a commission in the USAF through the Reserve Officer Training Corps program and entered the Air Force on active duty in July 1981. His first assignment as an Air Force Officer was to the 1000th Satellite Operations Group at Offutt AFB, Nebraska. His duty was as a System Integration Engineer, with responsibility for testing, integrating, and evaluating system upgrades for the command/control and telemetry processing systems in the Defense Meteorological Satellite Program; including telemetry analysis, communication, data base and retrieval systems. He left Nebraska when assigned to the Air Force Institute of Technology, School of Engineering, at Wright-Patterson AFB, Ohio in May of 1985.

Permanent address: [redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A177 652

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GE/ENG/86D-7	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/EN	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): see Box 19

12. PERSONAL AUTHOR(S): Miller, Dennis Max-David, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	December 1986	124

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software Science, Compiler performance, Ada, Compilers, Compiler Evaluation, Halstead
09	02		
05	10		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: Application of Halstead's Timing Model to Predict the Compilation Time of Ada Compilers

Approved for public release: IAW AFR 190-1.
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

Thesis Chairman: Wade H. Shaw, Jr., Ph.D.
Captain, U.S. Army
Assistant Professor of Electrical Engineering

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Wade H. Shaw Jr., PhD, CPT, U.S. Army	(513) 255-3576	AFIT/EN

DD Form 1473, JUN 86 — Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

With the development of Ada, the official programming language of the DoD, methods are needed to validate and evaluate various Ada compilers to determine if the compilers meet the DoD requirements. This investigation introduces a new tool using Halstead's Software Science theory to predict compile time and to evaluate the efficiency of alternative Ada compilers.

The analysis was accomplished by selecting a model based on Halstead's time equation. Once the model was established, programs from a benchmark test suite were used to evaluate the predictive power of the model and to develop a performance index for comparisons.

The results suggest that the compiler model is useful in predicting compile time, but of more importance, it is useful in the development of a performance index. The study shows that the average compilation time is not a good measure for comparing performance rates. Therefore, with further research, the compiler model may be a useful tool for software analysts.